

# Regular Substitution Sets: A Means of Controlling E-Unification

Jochen Burghardt

GMD Berlin,  
jochen@first.gmd.de

<http://www.first.gmd.de/persons/Burghardt.Jochen.html>

arXiv:1404.1201v1 [cs.FL] 4 Apr 2014

Technical Report  
Arbeitspapiere der GMD 926  
July 1995  
ISSN 0723-0508

GMD – Forschungszentrum  
Informationstechnik GmbH  
D-53754 Sankt Augustin  
Tel. \*49-2241-14-0  
Fax \*49-2241-14-2618  
Telex 889469 gmd d  
<http://www.gmd.de>

**Abstract.** A method for selecting solution constructors in narrowing is presented. The method is based on a sort discipline that describes regular sets of ground constructor terms as sorts. It is extended to cope with regular sets of ground substitutions, thus allowing different sorts to be computed for terms with different variable bindings. An algorithm for *computing* signatures of equationally defined functions is given that allows potentially infinite overloading. Applications to formal program development are sketched.



## 1 Motivation

Solving equations by narrowing has important applications, e.g. in the area of formal software development. However, the usual narrowing strategies are only able to restrict the set of application positions<sup>1</sup>. Ordered paramodulation [2] is able to provide a succession in which the defining equations have to be selected, but it cannot guarantee that an appropriate one is selected first. Bockmayr [3] has shown that, under certain general conditions, narrowing strategies essentially enumerate the whole term universe rather than specifically selecting the appropriate equations of a defined function to narrow with or the appropriate constructor to insert into the solution. In this paper, we present an approach for restricting the set of applicable defining equations in a narrowing step that is based on the dynamic computation of function signatures, rather than their declaration by a user.

The main idea is as follows<sup>2</sup>: As e.g. in [7], we distinguish between constructors and equationally defined functions; each well-defined ground term can be reduced to a ground constructor term, viz. its unique normal form. For a term  $v$ , let  $V$  be the set of all possible values of  $v$ , i.e., the set of all normal forms of admitted ground constructor instances of  $v$ . Then, a goal equation  $v_1 = v_2$  cannot be solved if  $V_1 \cap V_2 = \{\}$ ; in this case, it can be pruned from the search space of narrowing. Unfortunately,  $V_1$  and  $V_2$  are undecidable in general; to overcome this problem, we will define computable upper approximations  $\overline{V}_1 \supset V_1$  and  $\overline{V}_2 \supset V_2$ , respectively, and base the pruning decision on the consideration of  $\overline{V}_1 \cap \overline{V}_2$ .

To this end, we provide a framework of “extended sorts” to describe infinite sets of ground constructor terms like  $\overline{V}$  in a closed form, which is based on regular tree grammars (e.g. [13]). It is essential that extended sorts are closed wrt. intersection and that their inhabitation can be decided in order to conduct the above disjointness test. Moreover, set equality and subsort property can be decided, and  $\overline{V} = V$  always holds if  $v$  is a constructor term.

An algorithm for computing the extended sort  $\overline{V}$  from a term  $v$  is presented. In terms of conventional order-sorted rewriting, we thereby achieve potentially infinite overloading, since for an arbitrary input sort  $S$  we can *compute* a signature  $f : S \rightarrow \overline{f[S]}$  rather than being restricted to a few user-defined signatures which are generally too coarse for the disjointness test to be successfully applied. It is clear that the impact of this test on search-space reduction depends on the expressiveness of the sort framework and on the quality of signature approximation.

Consider, for example, the theory comprising equations  $a.$  to  $i.$  in Fig. 15. When trying to solve a goal equation like  $val(x) = s^5(0)$  wrt. this theory, conventional strategies are unable to decide which of the equations  $g.$ ,  $h.$ ,  $i.$  is to be used for a first narrowing step. Narrowing (at root position) with equation  $g.$ ,  $h.$ , and  $i.$  results in the new goal equations  $0 = s^5(0)$ ,  $dup(val(x')) = s^5(0)$ , and  $s(dup(val(x'))) = s^5(0)$ , respectively. While the first one is obviously false, the unsatisfiability of the second one can be detected as our algorithm computes the sort of its left-hand side as *Even* and recognizes that this is disjoint from its right-hand side’s sort,  $\{s^5(0)\}$ ; similarly, the third one is considered to be “possibly satisfiable” by the disjointness test. Hence, narrowing only makes sense with equation  $i.$ , and any solution to the above goal equation must take the form  $x = x' :: i.$  In Sect. 7 and App. B, examples of the pruning of infinite search-tree branches are given. Note that if a user were to declare the signatures  $+: Nat \times Nat \rightarrow Nat$ ,  $dup: Nat \rightarrow Nat$ , and  $val: Bin \rightarrow Nat$ , the disjointness test would allow narrowing with equations  $h.$  and  $i.$  In more complicated applications, a user cannot know in advance which signatures might become essential to disjointness tests in the course of the narrowing proof.

This example also shows that it is important to consider variable bindings during the computation of a term’s sort in order to get good approximations. For example, when computing a signature for  $dup$ ,

<sup>1</sup> Cf. e.g. the mathematical definition of the notion of strategy in [6].

<sup>2</sup> Notations and naming conventions are consistent with Def. 1 below.

the term  $x + x$  should be assigned the sort *Even*, whereas  $x + y$  can only be assigned *Nat*, assuming that  $x$  and  $y$  range over *Nat*. In conventional order-sorted approaches, the mapping from a term to its sort is usually a homomorphic extension of the sort assignment of variables, thus necessarily ignoring variable bindings, e.g.:

$$\begin{aligned} & \text{sortof}(x + x) \\ &= \text{get\_range\_from\_signatures}(+, \text{sortof}(x), \text{sortof}(x)) \\ &= \text{get\_range\_from\_signatures}(+, \text{sortof}(x), \text{sortof}(y)) \\ &= \text{sortof}(x + y). \end{aligned}$$

Instead, we use infinite sets of ground substitutions to denote sorts of variables, e.g.  $\{[x := s^i(0), y := s^j(0)] \mid i, j \in \mathbb{N}\}$  to indicate that  $x$  and  $y$  range over *Nat*. The mapping from a term to its set of possible values can then be achieved by applying each element of the substitution set, e.g.:

$$\begin{aligned} & \{[x := s^i(0)] \mid i \in \mathbb{N}\} (x + x) \\ &= \{[x := s^i(0)] (x + x) \mid i \in \mathbb{N}\} \\ &= \{s^i(0) + s^i(0) \mid i \in \mathbb{N}\}. \end{aligned}$$

Similarly,  $\{[x := s^i(0), y := s^j(0)] \mid i, j \in \mathbb{N}\} (x + y) = \{s^i(0) + s^j(0) \mid i, j \in \mathbb{N}\}$ . Both sets are different, hence the chance of finding different approximations for them within our extended sort framework is not forfeited<sup>3</sup>. In Fig. 13, we show that *Even* can in fact be obtained as the sort of  $x + x$ ; obtaining *Nat* for  $x + y$  is similar.

In order to have finite descriptions of such ground substitution sets, we express ground substitutions as ground constructor terms (“t-substitutions”) in a lifted algebra, allowing sets of them to be treated as tree languages (“t-sets”), and, in particular, to be described by regular tree grammars (“regular t-sets”). Regular t-sets can also express simple relations between distinct variables, allowing e.g. the representation of certain conditional equations by unconditional ones.

We provide a new class of tree languages, called “extended sorts”, which can be described by applying substitutions from a regular t-set  $\sigma$  to an arbitrary constructor term  $u$  with  $\text{vars}(u) \subset \text{dom}(\sigma)$ . In this way, the set of ground-constructor instances of an arbitrary constructor term can be expressed as an extended sort.

Regular string languages have been used e.g. by Mishra [11] as a basis for sort inference on Horn clauses. Owing to the restriction to *string* languages describing admissible paths in term trees, he is only able to express infinite sets that are closed wrt. all constructors; e.g. the set of all lists of naturals containing at least one 0 cannot be modeled. Comon [5] uses regular tree languages to describe sets of ground constructor terms as sorts, and the corresponding automaton constructions to implement sort operations. He provides a transformation system to decide first-order formulas with equality and sort membership as the only predicates. He shows the decision of inductive reducibility as an application. However, he does not consider equationally defined functions, e.g.  $(\forall x, y \ x+y = y+x) \rightarrow 0+1 = 1+0$  reduces to  $(\forall x, y \ x+y = y+x) \rightarrow \text{false}$  in his calculus.

Uribe [15] provides a unification algorithm for order-sorted terms in the presence of semilinear term declarations. The set of all ground constructor instances of a constructor term can then be described by a regular tree automaton with equality tests for direct subterms; allowing equality tests for arbitrary subterms makes the disjointness of two tree languages undecidable [14]. In our approach, arbitrary equality constraints may be imposed on subterms up to a fixed finite depth, whereas below that depth no equality constraints are allowed at all. Antimirov [1] suggested allowing regular t-sets

<sup>3</sup> Schmidt-Schauß [12] admits “term declarations”, allowing the user to declare different sorts for terms with different bindings. In our approach, however, the sorts are to be computed automatically.

with equality tests in extended sorts, thus extending the class of describable tree languages. This approach still remains to be investigated.

This paper is organized as follows. After a short introduction on regular sorts in Sect. 2, regular substitution sets and extended sorts are presented in Sect. 3 – 5. In Sect. 6, the algorithm for computing signatures of equationally defined functions is given. It is shown that an unsorted root-narrowing calculus from [9] remains complete if extended by appropriate sort restrictions. Section 7 sketches the application of narrowing to synthesize programs from formal specifications. Appendices A and B contain two case studies in program synthesis. For a full version including all proofs, see [4].

## 2 Regular Sorts

### Definition 1.

Let  $\mathcal{V}$  be a countable set of variables,  $\mathcal{CR}$  a finite set of term constructor symbols, each with fixed arity,  $\mathcal{F}$  a finite set of symbols for non-constructor functions, and  $\mathcal{S}$  a countable set of sort names. Let  $ar(g)$  denote the arity of a function symbol  $g$ . For a set<sup>4</sup> of symbols  $X \subset \mathcal{V} \cup \mathcal{CR} \cup \mathcal{F} \cup \mathcal{S}$ , let  $\mathcal{T}_X$  be the set of terms formed of symbols from  $X$ ; we abbreviate  $\mathcal{T}_{X \cup Y}$  to  $\mathcal{T}_{X,Y}$ . For example, the elements of  $\mathcal{T}_{\mathcal{CR}}$ ,  $\mathcal{T}_{\mathcal{CR},\mathcal{V}}$ , and  $\mathcal{T}_{\mathcal{CR},\mathcal{F},\mathcal{V}}$  are called ground constructor terms, constructor terms, and terms, respectively; the set  $\mathcal{T}_{\mathcal{CR},\mathcal{F},\mathcal{V},\mathcal{S}}$  is introduced in Sect. 6 for technical reasons. Let identifiers like  $u, u', u_i, \dots$  always denote members of  $\mathcal{T}_{\mathcal{CR},\mathcal{V}}$ ; similarly,  $v \in \mathcal{T}_{\mathcal{CR},\mathcal{F},\mathcal{V}}$ ,  $w \in \mathcal{T}_{\mathcal{CR},\mathcal{F},\mathcal{V},\mathcal{S}}$ ,  $x, y, z \in \mathcal{V}$ ,  $f \in \mathcal{F}$ ,  $g \in \mathcal{F} \cup \mathcal{CR}$ ,  $cr \in \mathcal{CR}$ , and  $S \in \mathcal{S}$ .

**Definition 2.**  $\langle v_1, \dots, v_n \rangle$  denotes an  $n$ -tuple,  $\langle v_i \mid p(v_i), i = 1, \dots, n \rangle$  denotes a tuple containing each  $v_i$  such that  $p(v_i)$  holds. We assume the existence of at least one nullary (e.g. *nil*) and one binary constructor (e.g. *cons*), so we can model arbitrary tuples as constructor terms. To improve readability, we sometimes write the application of a unary function  $f$  to its argument  $x$  as  $f \cdot x$ ;  $x^S$  stands, in the following, for the variable or constant  $x$  of sort  $S$ . We define the elementwise extension of a function  $f : A \rightarrow B$  to a set  $A' \subset A$  by  $f[A'] := \{f(a) \mid a \in A'\}$ .  $A \times B$  denotes the Cartesian product of sets  $A$  and  $B$ . For a finite set  $A$ , we denote its cardinality by  $\#A$ . We tacitly extend notations like  $\bigcup_{i=1}^n A_i$  to several binary operators defined in this paper, e.g.  $\big|_{i=1}^n S_i := S_1 \mid \dots \mid S_n$ .

**Definition 3.** Let  $vars(v_1, \dots, v_n)$  denote the set of variables occurring in any of the terms  $v_i$ . A term is called linear if it contains no multiple occurrences of the same variable; it is called pseudo-linear, if any two occurrences of the same variable are at the same depth; it is called semilinear if, for any two occurrences of the same variable, the lists of function symbols on each path from the root to an occurrence are equal. We write  $v_1 \triangleleft v_2$  to express that  $v_1$  is a proper subterm of  $v_2$ ; we write  $v_1 \triangleleft v_2$  for  $v_1 \triangleleft v_2 \vee v_1 = v_2$ . The depth of a position in a term is its distance from the root. We distinguish between “ordinary” substitutions, defined as usual (denoted by  $\beta, \gamma, \dots$ ), and “t-substitutions”, defined as constructor terms in Sect. 3, and denoted by  $\sigma', \tau', \dots$ . Application of a substitution  $\beta$  to a term  $v$  is written in prefix form, i.e.  $\beta v$ . For an ordinary substitution  $\beta$ , let  $dom(\beta) := \{x \in \mathcal{V} \mid \beta x \neq x\}$ , and  $ran(\beta) := \bigcup_{x \in dom(\beta)} vars(\beta x)$ .  $[x_1 := v_1, \dots, x_n := v_n]$  denotes the substitution that maps each  $x_i$  to  $v_i$ .  $\beta|_V$  denotes the domain restriction of  $\beta$  to a set  $V$  of variables. We assume all substitutions to be idempotent. If  $\beta_1$  and  $\beta_2$  agree on the intersection of their domains,  $\beta_1 \circ \beta_2$  denotes a “parallel composition” of them, i.e.

<sup>4</sup> “ $\subset$ ” denotes subset or equality, “ $\subsetneq$ ” denotes proper subset.

$$(\beta_1 \circ \beta_2)(x) := \begin{cases} \beta_1 x & \text{if } x \in \text{dom}(\beta_1) \\ \beta_2 x & \text{if } x \in \text{dom}(\beta_2) \end{cases}$$

$\beta_1 \circ \beta_2$  is undefined if  $\beta_1$  and  $\beta_2$  do not agree on  $\text{dom}(\beta_1) \cap \text{dom}(\beta_2)$ . A substitution  $\beta$  is called linear if the term  $\langle \beta x_1, \dots, \beta x_n \rangle$  is linear, where  $\{x_1, \dots, x_n\} = \text{dom}(\beta)$ ; similarly,  $\beta$  is called pseudolinear if  $\langle \beta x_1, \dots, \beta x_n \rangle$  is pseudolinear. We use the common notions of renaming substitution and most general unifier  $\beta = \text{mgu}(v_1, v_2)$ , however, we will additionally assume that  $v_1$  and  $v_2$  have disjoint variables and write  $\beta$  as  $\beta_1 \circ \beta_2$  with  $\text{dom}(\beta_1) = \text{vars}(v_1)$  and  $\text{dom}(\beta_2) = \text{vars}(v_2)$ .  $\text{mgu}$  is tacitly extended to finite sets of terms.

We follow the approach of [4] in describing regular sets of ground constructor terms as fixed points of sort equations, which is equivalent to the approach using finite tree automata [5], but provides a unique methodology for algorithms and proofs.

**Definition 4.** We allow sort definitions of the following syntax:

$$\begin{aligned} \text{SortName} &\doteq \text{SortName} \mid \dots \mid \text{SortName}, \\ \text{SortName} &\doteq \text{Constructor}(\text{SortName}, \dots, \text{SortName}) \end{aligned}$$

Let  $\dot{<}$  be the transitive closure of the relation  $S_i \dot{<} S \Leftrightarrow S \doteq S_1 \mid \dots \mid S_n$ . We admit finite systems of sort definitions such that  $\dot{<}$  is an irreflexive partial, hence well-founded, order. For example, the sort system consisting of  $A \doteq B$  and  $B \doteq A$  is forbidden. Each occurring sort name has to be defined. In examples, we generally use arbitrary sort expressions built from sort names, constructors, and “|” on the right-hand side of a sort definition. Any such sort system can be transformed to meet the above requirements while maintaining the least-fixed-point semantics given below.

For example, consider the sort definition  $\text{Bin} \doteq \text{nil} \mid \text{Bin}::o \mid \text{Bin}::i$  from Fig. 15 on page 40, which denotes the lists of binary digits, where “o” denotes zero, “i” denotes one, and  $::$  is an infix-*snoc*, i.e. reversed cons. The sort definition can be transformed into the corresponding definitions shown in Fig. 1, which obey Def. 4, by introducing new auxiliary sort names  $\text{Nil}$ ,  $\text{Bino}$ ,  $\text{Bini}$ ,  $O$ , and  $I$ .

Let  $X$  be an arbitrary mapping from sort names  $S$  to subsets  $S^X$  of  $\mathcal{T}_{\mathcal{CR}}$ .  $X$  is extended to sort expressions as follows:

$$\begin{aligned} (S_1 \mid S_2)^X &= S_1^X \cup S_2^X \\ \text{cr}(S_1, \dots, S_n)^X &= \text{cr}[S_1^X \times \dots \times S_n^X] \\ \text{cr}^X &= \{\text{cr}\} \end{aligned}$$

We say that  $X_1 \subset X_2$  if  $S^{X_1} \subset S^{X_2}$  for all sort names  $S$ . According to Thm. 5 below, for each admitted system of sort definition there exists exactly one mapping  $M$ , such that  $S^M = S'^M$  for each sort definition  $S \doteq S'$ . The semantics of a sort expression  $S$  is then defined as  $S^M$ .

**Theorem 5.** Each admitted system of sort definitions has exactly one fixed point.

*Proof.* If  $M$  and  $M'$  are fixed points of the sort definitions, use induction on the the lexicographic combination of  $\dot{<}$  and  $\dot{<}$  to show  $\forall u \forall S \quad u \in S^M \Rightarrow u \in S^{M'}$ .

**Definition 6.** For a sort name  $S$ , let  $\text{use}(S)$  denote the set of all sort names that occur directly or indirectly in the definition of  $S$ . For example,  $\text{use}(\text{Bin}) = \{O, I, \text{Bin}, \text{Bino}, \text{Bini}\}$ , cf. Fig. 1 and 2.

A subset  $T \subset \mathcal{T}_{\mathcal{CR}}$  is called regular if a system of sort definitions exists, such that  $T = S^M$  for some sort expression  $S$ . Note that  $u^M = \{u\}$  for all  $u \in \mathcal{T}_{\mathcal{CR}}$ , e.g.,  $s(0)^M = \{s(0)\}$ . The empty sort is denoted by  $\perp$ ; it can be defined e.g. by  $\perp \doteq s(\perp)$ . The uniqueness of fixed points validates the following induction principle, which is used in almost all correctness proofs of sort algorithms, cf. Alg. 10, 11, 37, 40, 41, 42, and 47.

**Theorem 7.** Let  $p$  be a family of unary predicates, indexed over the set of all defined sort names. Show for each defined sort name  $S$ :

$$\begin{array}{ll} \forall u \in \mathcal{T}_{\mathcal{CR}} \quad p_S(u) \leftrightarrow p_{S_1}(u) \vee \dots \vee p_{S_n}(u) & \text{if } S \doteq S_1 \mid \dots \mid S_n \\ \forall u \in \mathcal{T}_{\mathcal{CR}} \quad p_S(u) \leftrightarrow \exists u_1, \dots, u_n \in \mathcal{T}_{\mathcal{CR}} \quad u = cr(u_1, \dots, u_n) & \\ \quad \quad \quad \wedge p_{S_1}(u_1) \wedge \dots \wedge p_{S_n}(u_n) & \text{if } S \doteq cr(S_1, \dots, S_n) \\ \forall u \in \mathcal{T}_{\mathcal{CR}} \quad p_S(u) \leftrightarrow u = cr & \text{if } S \doteq cr \end{array}$$

Then,  $\forall u \in \mathcal{T}_{\mathcal{CR}} \quad u \in S^M \leftrightarrow p_S(u)$  holds for each defined sort name  $S$ .

*Proof.* The mapping  $S \mapsto \{u \in \mathcal{T}_{\mathcal{CR}} \mid p_S(u)\}$  is a fixed point of the sort definitions, hence the only one by Thm. 5.

**Theorem 8.** Let  $p$  be a family of unary predicates, indexed over the set of all defined sort names. Show for each defined sort name  $S$ :

$$\begin{array}{lll} \forall u \in \mathcal{T}_{\mathcal{CR}} & p_S(u) & \leftarrow p_{S_1}(u) \vee \dots \vee p_{S_n}(u) \quad \text{if } S \doteq S_1 \mid \dots \mid S_n \\ \forall u_1, \dots, u_n \in \mathcal{T}_{\mathcal{CR}} & p_S(cr(u_1, \dots, u_n)) & \leftarrow p_{S_1}(u_1) \wedge \dots \wedge p_{S_n}(u_n) \quad \text{if } S \doteq cr(S_1, \dots, S_n) \\ \forall u \in \mathcal{T}_{\mathcal{CR}} & p_S(cr) & \quad \text{if } S \doteq cr \end{array}$$

Then,  $\forall u \in S^M \quad p_S(u)$  holds for each defined sort name  $S$ .

*Proof.* Use Scott's fixed-point induction. The Thm. remains valid even if  $\dot{<}$  is not irreflexive.

**Corollary 9.** For each sort name  $S$ , we provide the following structural induction principle: show for each sort definition  $S' \doteq cr(S'_1, \dots, S'_n)$  such that  $S' \in use(S)$ , and  $S'^M \subset S^M$ :

$$\forall u_1, \dots, u_n \in \mathcal{T}_{\mathcal{CR}} \quad \left( \bigwedge_{\substack{i=1 \dots n \\ S'_i{}^M \subset S^M}} u_i \in S'_i{}^M \wedge p(u_i) \right) \longrightarrow p(cr(u_1, \dots, u_n))$$

Then,  $\forall u \in S^M \quad p(u)$  holds.  $S'^M \subset S^M$  can be decided using Alg. 12 below.

*Proof.* Use Thm. 8 with  $p_{S'}(u) :\Leftrightarrow \begin{cases} p(u) & \text{if } S' \in use(S) \text{ and } S'^M \subset S^M \\ true & \text{else} \end{cases}$ .

Figure 2 shows an induction principle for sort *Bin* from Fig. 1, using Cor. 9. Algorithms for computing the intersection and the relative complement of two regular sorts, as well as for deciding the inhabitation of a sort, and thus of the subsort and sort equivalence property, are given below. They consist essentially of distributivity rules, constructor-matching rules, and loop-checking rules. The latter stop the algorithm, when it calls itself recursively with the same arguments, and generate a corresponding new recursive sort definition.



$Bin$	$\doteq Nil \mid Bino \mid Bini$	binary numbers, i.e. lists of binary digits
$Bin^0$	$\doteq Nil \mid Bino^0$	binary numbers that contain no ones
$Bin^{n+1}$	$\doteq Nil \mid Bino^{n+1} \mid Bini^n$	for $n \geq 0$ binary numbers that contain at most $n + 1$ ones
$Bin_0$	$\doteq Bin$	binary numbers that contain at least 0 ones
$Bin_{n+1}$	$\doteq Bino_{n+1} \mid Bini_n$	for $n \geq 0$ binary numbers that contain at least $n + 1$ ones
$Bino$	$\doteq Bin :: O$	binary numbers with a trailing zero
$Bini$	$\doteq Bin :: I$	binary numbers with a trailing one
$Bino^n$	$\doteq Bin^n :: O$	
$Bini^n$	$\doteq Bin^n :: I$	
$Bino_n$	$\doteq Bin_n :: O$	
$Bini_n$	$\doteq Bin_n :: I$	
$Nil$	$\doteq nil$	empty list
$O$	$\doteq o$	zero-digit
$I$	$\doteq i$	one-digit

**Fig. 1.** Examples of sort definitions

$S'^M \subset Bin^M$		
$S' \doteq cr(\dots) \mid$		
$S' \in use(Bin) \downarrow \downarrow$		
	$O$	$+-$
	$I$	$+-$
	$Bin$	$-+$
$p(nil)$	$Nil$	$++$
$\forall u_1 \ u_1 \in Bin^M \wedge p(u_1) \rightarrow p(u_1 :: o)$	$Bino$	$++$
$\forall u_1 \ u_1 \in Bin^M \wedge p(u_1) \rightarrow p(u_1 :: i)$	$Bini$	$++$
$\forall u \in Bin^M \ p(u)$		

**Fig. 2.** Induction principle for sort  $Bin$

$\inf(Bin^2, Bin_1) = Sort_1$	$\doteq \inf(Nil, Bin_1) \mid \inf(Bino^2, Bin_1) \mid \inf(Bini^1, Bin_1)$	by 2.
$\inf(Nil, Bin_1) = Sort_2$	$\doteq \inf(Nil, Bino_1) \mid \inf(Nil, Bini_0)$	by 3.
$\inf(Bino^2, Bin_1) = Sort_3$	$\doteq \inf(Bino^2, Bino_1) \mid \inf(Bino^2, Bini_0)$	by 3.
$\inf(Bini^1, Bin_1) = Sort_4$	$\doteq \inf(Bini^1, Bino_1) \mid \inf(Bini^1, Bini_0)$	by 3.
$\inf(Nil, Bino_1) = Sort_5$	$\doteq \perp$	by 5.
$\inf(Nil, Bini_0) = Sort_6$	$\doteq \perp$	by 5.
$\inf(Bino^2, Bino_1) = Sort_7$	$\doteq \inf(Bin^2, Bin_1) :: \inf(O, O)$	by 4.
$\inf(Bino^2, Bini_0) = Sort_8$	$\doteq \inf(Bin^2, Bin_0) :: \inf(O, I)$	by 4.
$\inf(Bini^1, Bino_1) = Sort_9$	$\doteq \inf(Bin^1, Bin_1) :: \inf(I, O)$	by 4.
$\inf(Bini^1, Bini_0) = Sort_{10}$	$\doteq \inf(Bin^1, Bin_0) :: \inf(I, I)$	by 4.
$\inf(Bin^2, Bin_1) = Sort_1$		by 1.
$\inf(O, O) = Sort_{11}$	$\doteq O$	by 4.
$\inf(Bin^2, Bin_0) = Sort_{12}$	$\doteq Bin^2$	by similar computations
$\inf(O, I) = Sort_{13}$	$\doteq \perp$	by 5.
$\inf(Bin^1, Bin_1) = Sort_{14}$	$\doteq \dots$	by similar computations
$\inf(I, O) = Sort_{15}$	$\doteq \perp$	by 5.
$\inf(Bin^1, Bin_0) = Sort_{16}$	$\doteq \inf(Bin^1, Bin)$	by 3.
$\inf(I, I) = Sort_{17}$	$\doteq I$	by 4.
$\inf(Bin^1, Bin) = Bin^1$		by similar computations
Hence,		
$Sort_1 \doteq Sort_2 \mid Sort_3 \mid Sort_4,$		
$Sort_2 \doteq Sort_5 \mid Sort_6,$		
$Sort_3 \doteq Sort_7 \mid Sort_8,$		
$Sort_4 \doteq Sort_9 \mid Sort_{10},$		
$Sort_5 \doteq \perp$		
$Sort_6 \doteq \perp$		
$Sort_7 \doteq Sort_1 :: Sort_{11},$		
$Sort_8 \doteq Sort_{12} :: Sort_{13},$		
$Sort_9 \doteq Sort_{14} :: Sort_{15},$		
$Sort_{10} \doteq Sort_{16} :: Sort_{17},$		
$Sort_{16} \doteq Bin^1$		
$Sort_{11} \doteq O$		
$Sort_{12} = Bin^2$		
$Sort_{13} \doteq \perp$		
$Sort_{14} \doteq \dots$		
$Sort_{15} \doteq \perp$		
$Sort_{17} \doteq I$		

**Fig. 3.** Example computation of sort infimum

**Algorithm 10.** The following algorithm computes the intersection of two regular sorts. Let  $S_1$  and  $S_2$  be sort names, let  $S$  be a new sort name. Define  $\text{inf}(S_1, S_2) = S$ , where a new sort definition is introduced for  $S$ :

1. If  $\text{inf}(S_1, S_2)$  has already been called earlier,  $S$  is already defined (loop check).
2. Else, if  $S_1 \doteq S_{11} \mid \dots \mid S_{1n}$ , define  $S \doteq \text{inf}(S_{11}, S_2) \mid \dots \mid \text{inf}(S_{1n}, S_2)$
3. Else, if  $S_2 \doteq S_{21} \mid \dots \mid S_{2n}$ , define  $S \doteq \text{inf}(S_1, S_{21}) \mid \dots \mid \text{inf}(S_1, S_{2n})$
4. Else, if  $S_1 \doteq \text{cr}(S_{11}, \dots, S_{1n})$  and  $S_2 \doteq \text{cr}(S_{21}, \dots, S_{2n})$ ,  
define  $S \doteq \text{cr}(\text{inf}(S_{11}, S_{21}), \dots, \text{inf}(S_{1n}, S_{2n}))$
5. Else, define  $S \doteq \perp$

Using Thm. 7 with  $p_S(u) :\Leftrightarrow \begin{cases} u \in S_1^M \cap S_2^M & \text{if } S = \text{inf}(S_1, S_2) \\ u \in S^M & \text{else} \end{cases}$

it can be shown that  $\text{inf}(S_1, S_2)^M = S_1^M \cap S_2^M$ . The *else*-case in the definition of  $p_S(u)$  causes only trivial proof obligations; in later applications of Thm. 7, it will be tacitly omitted for the sake of brevity. The algorithm obviously needs at most  $\#use(S_1) * \#use(S_2)$  recursive calls to compute  $\text{inf}(S_1, S_2)$ .

**Algorithm 11.** The following algorithm computes the relative complement of two regular sorts. For technical reasons, the second argument may be an arbitrary union of sort names. Let  $S_1, \dots, S_m$  be sort names, let  $S$  be a new sort name. Define  $\text{diff}(S_1, S_2 \mid \dots \mid S_m) = S$ , where a new sort definition is introduced for  $S$ :

1. If  $\text{diff}(S_1, S_2 \mid \dots \mid S_m)$  has already been called earlier,  $S$  is already defined (loop check).
2. If  $S_1 \doteq S_{11} \mid \dots \mid S_{1n}$ , define  $S \doteq \text{diff}(S_{11}, S_2 \mid \dots \mid S_m) \mid \dots \mid \text{diff}(S_{1n}, S_2 \mid \dots \mid S_m)$
3. If  $S_i \doteq S_{i1} \mid \dots \mid S_{in}$  for  $2 \leq i \leq m$ , define  $S \doteq \text{diff}(S_1, S_2 \mid \dots \mid S_{i-1} \mid S_{i+1} \mid \dots \mid S_m \mid S_{i1} \mid \dots \mid S_{in})$
4. If  $S_1 \doteq \text{cr}(S_{11}, \dots, S_{1n}) \dots, S_m \doteq \text{cr}(S_{m1}, \dots, S_{mn})$ , with  $n > 0$ ,  
let  $S_{l_1, \dots, l_m}$  be a new sort name for each  $l_1, \dots, l_m \in \{1, \dots, n\}$ ,  
define  $S \doteq \bigsqcup_{l_1=1}^n \dots \bigsqcup_{l_m=1}^n S_{l_1, \dots, l_m}$   
and  $S_{l_1, \dots, l_m} \doteq \text{cr}(\text{diff}(S_{11}, \bigsqcup_{j \geq 2, l_j=1} S_{j1}), \dots, \text{diff}(S_{1n}, \bigsqcup_{j \geq 2, l_j=n} S_{jn}))$ .
5. If  $S_1 \doteq \text{cr}$ ,  $S_2 \doteq \text{cr}$  and  $m = 2$ , define  $S \doteq \perp$
6. If  $S_1 \doteq \text{cr}(\dots)$  and  $S_m \doteq \text{cr}'(\dots)$  with  $\text{cr} \neq \text{cr}'$  and  $m > 2$ ,  
define  $S \doteq \text{diff}(S_1, S_2 \mid \dots \mid S_{m-1})$
7. If  $S_1 \doteq \text{cr}(\dots)$  and  $S_m \doteq \text{cr}'(\dots)$  with  $\text{cr} \neq \text{cr}'$  and  $m = 2$ ,  
define  $S \doteq S_1$
8. If  $S_2 \doteq \perp$  and  $m = 2$ , define  $S \doteq S_1$ .

Using Thm. 7 with

$$\begin{aligned} p_S(u) &:\Leftrightarrow u \in S_1^M \setminus (S_2^M \cup \dots \cup S_m^M) && \text{if } S = \text{diff}(S_1, S_2) \text{ and} \\ p_{S_{l_1, \dots, l_m}}(u) &:\Leftrightarrow \exists u_1, \dots, u_n \ u = \text{cr}(u_1, \dots, u_n) \\ &\quad \wedge \bigwedge_{i=1}^n u_i \in (S_{1i}^M \setminus \bigcup_{j \geq 2, l_j=i} S_{ji}^M) && \text{if } S_{l_1, \dots, l_m} \text{ was defined in rule 4.,} \end{aligned}$$

it can be shown that  $\text{diff}(S_1, S_2 \mid \dots \mid S_m)^M = S_1^M \setminus (S_2^M \cup \dots \cup S_m^M)$ .

The algorithm needs at most  $\#use(S_1) * 2^{\#use(S_2)}$  recursive calls to compute  $\text{diff}(S_1, S_2)$ .

**Algorithm 12.** Let  $S$  be a sort name, define  $\text{inh}(S, \text{Occ}) = \langle A, B, C, D \rangle$  where  $A$  is a finite set of ground constructor terms,  $B \in \{\text{true}, \text{false}\}$ ,  $C, D$ , and  $\text{Occ}$  are finite sets of sort names, as follows:

$diff(Bin, Bin_1)$	$= Sort_{18} \doteq diff(Nil, Bin_1) \mid diff(Bino, Bin_1)$	by 2.
	$\mid diff(Bini, Bin_1)$	
$diff(Nil, Bin_1)$	$= Sort_{19} \doteq diff(Nil, Bino_1 \mid Bini_0)$	by 3.
$diff(Bino, Bin_1)$	$= Sort_{20} \doteq diff(Bino, Bino_1 \mid Bini_0)$	by 3.
$diff(Bini, Bin_1)$	$= Sort_{21} \doteq diff(Bini, Bino_1 \mid Bini_0)$	by 3.
$diff(Nil, Bino_1 \mid Bini_1)$	$= Sort_{22} \doteq diff(Nil, Bino_1)$	by 6.
$diff(Nil, Bino_1)$	$= Sort_{23} \doteq Nil$	by 7.
$diff(Bino, Bino_1 \mid Bini_0)$	$= Sort_{24} \doteq Sort_{25} \mid Sort_{26} \mid Sort_{27} \mid Sort_{28}$	by 4.
	$Sort_{25} \doteq diff(Bin, Bin_1 \mid Bino) :: diff(O, \perp)$	1,1
	$Sort_{26} \doteq diff(Bin, Bin_1) :: diff(O, I)$	1,2
	$Sort_{27} \doteq diff(Bin, Bino) :: diff(O, O)$	2,1
	$Sort_{28} \doteq diff(Bin, \perp) :: diff(O, O \mid I)$	2,2
$diff(Bini, Bino_1 \mid Bini_0)$	$= Sort_{29} \doteq Sort_{30} \mid Sort_{31} \mid Sort_{32} \mid Sort_{33}$	by 4.
	$Sort_{30} \doteq diff(Bin, Bin_1 \mid Bino) :: diff(I, \perp)$	1,1
	$Sort_{31} \doteq diff(Bin, Bin_1) :: diff(I, I)$	1,2
	$Sort_{32} \doteq diff(Bin, Bino) :: diff(I, O)$	2,1
	$Sort_{33} \doteq diff(Bin, \perp) :: diff(I, O \mid I)$	2,2
$diff(Bin, Bin_1 \mid Bino)$	$= \perp$	by similar computations
$diff(Bin, Bin_1)$	$= Sort_{18}$	by 1.
$diff(Bin, Bino)$	$= \perp$	by similar computations
$diff(Bin, \perp)$	$= Bin$	by 8.
$diff(O, \perp)$	$= O$	by 8.
$diff(O, I)$	$= O$	by 7.
$diff(O, O)$	$= \perp$	by 5.
$diff(O, O \mid I)$	$= \perp$	by 6.,5.
$diff(I, \perp)$	$= I$	by 8.
$diff(I, I)$	$= \perp$	by 5.
$diff(I, O)$	$= I$	by 7.
$diff(I, O \mid I)$	$= \perp$	by 6.,5.
Hence,		
$Sort_{18}$	$\doteq Sort_{19} \mid Sort_{20} \mid Sort_{21}$	
$Sort_{19}$	$\doteq Sort_{22}$	
$Sort_{20}$	$\doteq Sort_{24}$	
$Sort_{21}$	$\doteq Sort_{29}$	
$Sort_{22}$	$\doteq Sort_{23}$	
$Sort_{23}$	$\doteq Nil$	
$Sort_{24}$	$\doteq Sort_{25} \mid Sort_{26} \mid Sort_{27} \mid Sort_{28}$	
$Sort_{25}$	$\doteq \perp :: O$	
$Sort_{26}$	$\doteq Sort_{18} :: O$	
$Sort_{27}$	$\doteq \perp :: \perp$	
$Sort_{28}$	$\doteq Bin :: \perp$	
$Sort_{29}$	$\doteq Sort_{30} \mid Sort_{31} \mid Sort_{32} \mid Sort_{33}$	
$Sort_{30}$	$\doteq \perp :: I$	
$Sort_{31}$	$\doteq Sort_{18} :: \perp$	
$Sort_{32}$	$\doteq \perp :: I$	
$Sort_{33}$	$\doteq Bin :: \perp$	

**Fig. 4.** Example computation of sort difference

1. If  $S \in Occ$ , define  $inh(S, Occ) = \langle \{\}, false, \{S\}, \{S\} \rangle$
2. Else, if  $S \doteq S_1 \mid \dots \mid S_n$ , define  $inh(S, Occ) = \langle A_1 \cup \dots \cup A_n, B, C, D \rangle$
3. Else, if  $S \doteq cr(S_1, \dots, S_n)$ , define  $inh(S, Occ) = \langle cr[A_1 \times \dots \times A_n], B, C, D \rangle$
4. Else, if  $S \doteq cr$ , define  $inh(S, Occ) = \langle \{cr\}, false, \{S\}, \{\} \rangle$

where  $\langle A_i, B_i, C_i, D_i \rangle := inh(S_i, Occ \cup \{S\})$  for  $i = 1, \dots, n$ ,  $B := B_1 \vee \dots \vee B_n \vee S \in C_1 \cup \dots \cup C_n$ ,  $C := C_1 \cup \dots \cup C_n \cup \{S\}$ , and  $D := (D_1 \cup \dots \cup D_n) \setminus \{S\}$ .  $A$  is used to decide  $S^M \neq \{\}$ ,  $B$  is *true* if a loop occurs in the definition of  $S$ ,  $C$  is used to compute  $B$ , and  $D$  is used only for proof technical reasons and need not be computed in a practical implementation. Let  $inh(S, \{\}) = \langle A, B, C, D \rangle$ . Then,  $S^M \neq \{\}$  iff  $A \neq \{\}$ ;  $S^M$  finite iff  $B \Leftrightarrow false$ , and in this case  $A = S^M$ . The algorithm needs at most  $\#use(S) * 2^{\#use(S)}$  recursive calls to compute  $inh(S, \{\})$ . Define  $single(S) := inh(S, \{\}) = \langle \{u\}, false, C, D \rangle$  for some  $u, C, D$ .

*Proof.*

1. Use  $Occ_1 < Occ_2 : \Leftrightarrow Occ_2 \subsetneq Occ_1$  to show termination and complexity; note that  $Occ_1, Occ_2$  is bounded from above by the finite set  $use(S)$ .
2. Let  $inh(S, Occ) = \langle A, B, C, D \rangle$ , and  $E := \{u \mid \exists S' \in D, u' \in S'^M \ u' \triangleleft u\}$ ; all following statements are proven by induction on the computation tree of  $inh(S, Occ)$ .
3. Show  $D \subset Occ \cap C$ , hence  $E = \{\}$  if  $Occ = \{\}$ .
4. If  $B = false$ , show  $S^M \subset A \cup E$ .
5. Show  $A \subset S^M$ .
6. If  $A = \{\}$ , show  $S^M \subset E$  by induction on the computation tree, and (nested) induction on  $u \in S^M$ .
7. If  $Occ = \{\}$ , from 5. (" $\Leftarrow$ "), 3. and 6. (" $\Rightarrow$ ") follows  $S^M \neq \{\}$  iff  $A \neq \{\}$ .
8. Show  $B \Leftrightarrow false$  iff  $S$  contains no loops (iff  $S^M$  is finite by the pumping lemma).
9. If  $Occ = \{\}$ , from 5. (" $\subset$ "), 3. and 4. (" $\supset$ ") follows  $A = S^M$  if  $B \Leftrightarrow false$ .

Using the sort definitions from Fig. 1, Fig. 3 shows the computation of the intersection of  $Bin^2$  and  $Bin_1$  by Alg. 10; the result may be simplified to  $Sort_1 \doteq Sort_1 :: o \mid Bin^1 :: i$ , which uses the sloppy notation for sort definitions mentioned in Def. 4, and intuitively denotes all binary lists with one or two  $i$ -digits. Figure 4 shows the computation of the complement of  $Bin_1$  relative to  $Bin$  by Alg. 11; the result may be simplified to  $Sort_{18} \doteq nil \mid Sort_{18} :: o$  which is equivalent to  $Bin^0$ .

In [4], sort definitions may include "constraint formulas" which are not to be considered by the sort algorithms, but rather collected and passed to an external prover in which the sort algorithms are meant to be embedded. A sort definition (cf. Def. 4) may also have the form

$$SortName \doteq Constructor(Id : SortName, \dots, Id : SortName) \triangleleft Constraint(Id, \dots, Id),$$

with the semantics

$$(cr(i_1 : S_1, \dots, i_n : S_n) \triangleleft p(i_1, \dots, i_n))^M = \{cr(u_1, \dots, u_n) \mid u_1 \in S_1^M, \dots, u_n \in S_n^M, p(u_1, \dots, u_n)\};$$

and e.g. rule 4. of Alg. 10 has then the following form

$$\begin{aligned} &\text{If } S_1 \doteq cr(S_{11}, \dots, S_{1n}) \triangleleft ct_1, \text{ and } S_2 \doteq cr(S_{21}, \dots, S_{2n}) \triangleleft ct_2, \\ &\text{define } S \doteq cr(inf(S_{11}, S_{21}), \dots, inf(S_{1n}, S_{2n})) \triangleleft ct_1 \wedge ct_2. \end{aligned}$$

The same applies to Alg. 11. Algorithm 12 may yield a proper predicate  $B \notin \{true, false\}$  if a nontrivial constraint formula occurs above a loop in the sort definition. Constraint formulas are mentioned here only because they appear in App. B.

### 3 T-Substitutions

In this section, we apply the formalism from Sect. 2 to define possibly infinite regular sets of ground substitutions. We define suitable free constructors from which ground substitutions can be built as terms of a lifted algebra  $\mathcal{T}_{(\mathcal{V} \rightarrow \mathcal{CR})}^*$ . We call such terms t-substitutions. Note that the classical approach, constructing substitutions by functional composition from simple substitutions, cannot be used, since functional composition is not free but obeys e.g. the associativity law. We provide the necessary notions and properties of t-substitutions and of sets of them, called t-sets. All results in this section hold for arbitrary t-sets.

We first define suitable free constructors from which ground substitutions can be built as terms of a lifted algebra. Expressed informally, to build a substitution term corresponding to  $[x_1 := u_1, \dots, x_n := u_n]$  with  $u_i$  ground, we “overlay” the  $u_i$  to obtain the substitution term; on the right, an example is shown for  $[x := \text{cons}(0, \text{nil}), y := s(s(0))]$ .

$$\begin{array}{l} x := \text{cons} ( 0, \text{nil} ) \\ y := s ( s ( 0 ) ) \\ \hline \text{cons}_x s_y (0_x s_y (0_y), \text{nil}_x) \end{array}$$

**Definition 13.** Given a set  $V \subset \mathcal{V}$  of variables, define the constructors for t-substitutions with domain  $V$  as the set  $(V \rightarrow \mathcal{CR})$  of all total mappings from  $V$  to  $\mathcal{CR}$ . T-substitution constructors are denoted by  $\mathbf{cr}, \mathbf{cr}', \dots$ , the empty mapping by  $\varepsilon$ . Function application is written as  $\mathbf{cr}_x$ , the arity is defined as  $ar(\mathbf{cr}) := \max_{x \in \text{dom}(\mathbf{cr})} ar(\mathbf{cr}_x)$ . For  $V' \subset \mathcal{V}$ , let  $\mathbf{cr}|_{V'}$  denote the restriction of  $\mathbf{cr}$  to the variables in  $V'$ , i.e.  $\text{dom}(\mathbf{cr}|_{V'}) = V'$  and  $(\mathbf{cr}|_{V'})_x = \mathbf{cr}_x$  for all  $x \in V'$ . For  $\mathbf{cr}$  and  $\mathbf{cr}'$  such that  $\mathbf{cr}_x = \mathbf{cr}'_x$  for all  $x \in \text{dom}(\mathbf{cr}) \cap \text{dom}(\mathbf{cr}')$ , let  $\mathbf{cr} \diamond \mathbf{cr}'$  denote the “parallel composition” of  $\mathbf{cr}$  and  $\mathbf{cr}'$ , i.e.  $\text{dom}(\mathbf{cr} \diamond \mathbf{cr}') = \text{dom}(\mathbf{cr}) \cup \text{dom}(\mathbf{cr}')$ , and

$$(\mathbf{cr} \diamond \mathbf{cr}')_x = \begin{cases} \mathbf{cr}_x & \text{if } x \in \text{dom}(\mathbf{cr}) \\ \mathbf{cr}'_x & \text{if } x \in \text{dom}(\mathbf{cr}') \end{cases}.$$

Note that  $\mathbf{cr} \diamond \mathbf{cr}'$  is undefined if  $\mathbf{cr}$  and  $\mathbf{cr}'$  do not agree on their domain intersection.

**Example 14.** In examples, we write e.g.  $0_x s_y$  to denote the mapping  $(x \mapsto 0, y \mapsto s)$ . We have

$$\begin{aligned} 0_x s_y &\in (\{x, y\} \rightarrow \mathcal{CR}), \\ ar(0_x s_y) &= \max(0, 1) = 1, \\ (0_x s_y)|_{\{y\}} &= s_y, \text{ and} \\ (0_x s_y) \diamond (s_y \text{cons}_z) &= 0_x s_y \text{cons}_z. \end{aligned}$$

**Definition 15.** Once we have defined t-substitution constructors, we inherit the initial term algebra  $\mathcal{T}_{(V \rightarrow \mathcal{CR})}$  over them. However, we have to exclude some nonsense terms. Define the subset  $\mathcal{T}_{(V \rightarrow \mathcal{CR})}^* \subset \mathcal{T}_{(V \rightarrow \mathcal{CR})}$  of admissible t-substitutions with domain  $V$  as the least set such that

$$\begin{aligned} \mathbf{cr}(\sigma'_1, \dots, \sigma'_{ar(\mathbf{cr})}) &\in \mathcal{T}_{(V \rightarrow \mathcal{CR})}^* \\ \text{if } \mathbf{cr} &\in (V \rightarrow \mathcal{CR}) \text{ and } \sigma'_i \in \mathcal{T}_{(\{x \in V \mid ar(\mathbf{cr}_x) \geq i\} \rightarrow \mathcal{CR})}^* \text{ for } i = 1, \dots, ar(\mathbf{cr}). \end{aligned}$$

We denote t-substitutions by  $\sigma', \tau', \mu', \dots$ . Sets of t-substitutions are called t-sets and are denoted by  $\sigma, \tau, \mu, \dots$ .

T-substitutions, built as constructor terms:	
$s_x(s_x(0_x))$	$\triangleq [x := s(s(0))]$
$s_x s_y(0_x 0_y)$	$\triangleq [x := s(0), y := s(0)]$
$s_x 0_y(0_x)$	$\triangleq [x := s(0), y := 0]$

**Fig. 5.** Examples of t-substitutions

**Example 16.** Definition 15 implies that  $\mathbf{cr} \in \mathcal{T}_{(V \rightarrow \mathcal{CR})}^*$  if  $\mathbf{cr} \in (V \rightarrow \mathcal{CR})$  is a nullary t-substitution constructor. For example, we have  $0_y \in \mathcal{T}_{(\{y\} \rightarrow \mathcal{CR})}^*$ , and hence  $0_x s_y(0_y) \in \mathcal{T}_{(\{x,y\} \rightarrow \mathcal{CR})}^*$ , but neither  $0_y \in \mathcal{T}_{(\{x,y\} \rightarrow \mathcal{CR})}^*$ , nor  $0_x s_y(0_x 0_y) \in \mathcal{T}_{(V \rightarrow \mathcal{CR})}^*$  for any  $V$ . Figure 5 shows some more t-substitutions together with their intended semantics.

**Definition 17.** Since  $\mathcal{T}_{(V_1 \rightarrow \mathcal{CR})}^* \cap \mathcal{T}_{(V_2 \rightarrow \mathcal{CR})}^* = \{\}$  for  $V_1 \neq V_2$ , we may define  $\text{dom}(\sigma') := V$  iff  $\sigma' \in \mathcal{T}_{(V \rightarrow \mathcal{CR})}^*$ . Let  $(V \hookrightarrow \mathcal{CR})$  be the set of all partial mappings from  $V$  to  $\mathcal{CR}$ ; define the set of admissible t-substitutions with a subset of  $V$  as domain by  $\mathcal{T}_{(V \hookrightarrow \mathcal{CR})}^* := \bigcup_{V' \subset V, V' \text{ finite}} \mathcal{T}_{(V' \rightarrow \mathcal{CR})}^*$ .

**Definition 18.** Define the t-substitution application  $\sigma' u$  by

$$\sigma'(cr(u_1, \dots, u_k)) := cr[\sigma' u_1 \times \dots \times \sigma' u_k]$$

$$\sigma'(cr) := \{cr\}$$

$$(\mathbf{cr}(\sigma'_1, \dots, \sigma'_n))(x) := \mathbf{cr}_x[\sigma'_1 x \times \dots \times \sigma'_{ar(\mathbf{cr}_x)} x] \quad \text{if } x \in \text{dom}(\mathbf{cr}), n > 0$$

$$(\mathbf{cr})(x) := \{\mathbf{cr}_x\} \quad \text{if } x \in \text{dom}(\mathbf{cr}), n = 0$$

$$(\mathbf{cr}(\sigma'_1, \dots, \sigma'_n))(x) := \{\} \quad \text{if } x \notin \text{dom}(\mathbf{cr})$$

$\sigma' u$  yields a set with at most one ground constructor term. Application is extended elementwise to t-sets by  $\sigma u := \bigcup_{\sigma' \in \sigma} \sigma' u$ .

Note that, in contrast to an ordinary substitution  $\beta$ , a t-substitution  $\sigma'$  is undefined outside its domain, i.e. it returns the empty set. We have  $\varepsilon u = \{\}$  if  $u$  contains variables,  $\varepsilon u = u$  if  $u$  is ground, and always  $\perp u = \{\}$ .

**Lemma 19.**  $\sigma' = \tau'$  iff  $\sigma' x = \tau' x$  for all  $x \in \mathcal{V}$ ,

where “=” on the left-hand side denotes the syntactic equality in  $\mathcal{T}_{(V \hookrightarrow \mathcal{CR})}^*$ .

Although constructors may be written in different ways, e.g.  $0_x s_y = s_y 0_x$ , the initiality condition  $\mathbf{cr}(u_1, \dots, u_n) = \mathbf{cr}'(u'_1, \dots, u'_n) \Rightarrow \mathbf{cr} = \mathbf{cr}' \wedge n = n' \wedge u_1 = u'_1 \wedge \dots \wedge u_n = u'_n$  is satisfied in  $\mathcal{T}_{(V \hookrightarrow \mathcal{CR})}^*$ . The desired equivalence of term equality and function equality from Lemma 19 is the reason for restricting t-substitutions to a subset  $\mathcal{T}_{(V \hookrightarrow \mathcal{CR})}^*$  of the initial algebra  $\mathcal{T}_{(V \hookrightarrow \mathcal{CR})}$ , excluding nonsense terms like e.g.  $0_x s_y(0_x 0_y)$  and  $0_x s_y(0_x 1_y)$  which would contradict the initiality requirement.

**Lemma 20.**  $\mathcal{T}_{(V \hookrightarrow \mathcal{CR})}^*$  corresponds to the set of all ordinary ground substitutions in the following sense: For each  $\sigma'$  there exists a  $\beta$ , such that  $\sigma' u = \{\beta u\}$  whenever  $\text{vars}(u) \subset \text{dom}(\sigma')$ . Conversely, for each  $\beta$  there exists a  $\sigma'$  with the respective property; cf. Fig. 5 which shows some example correspondences.

*Proof.* Induction on  $\sigma'$  with  $\beta_{\mathbf{cr}(\sigma'_1, \dots, \sigma'_n)}(x) := \mathbf{cr}_x(\beta_{\sigma'_1} x, \dots, \beta_{\sigma'_n} x)$ .

Conversely: define  $\{\sigma'_\beta\} := \diamond_{x \in \text{dom}(\beta)} [x := \beta x]$ . Then,  $\sigma'_\beta u = \{\beta u\}$  whenever  $\text{vars}(u) \subset \text{dom}(\beta)$ .

**Definition 21.** Define the t-substitution restriction  $\sigma'|_V$  by  
 $\mathbf{cr}|_V$  as defined in Def. 13 if  $ar(\mathbf{cr}) = 0$   
 $(\mathbf{cr}(\sigma'_1, \dots, \sigma'_n))|_V := (\mathbf{cr}|_V)(\sigma'_1|_V, \dots, \sigma'_n|_V)$  if  $ar(\mathbf{cr}) > 0$  and  $ar(\mathbf{cr}|_V) = m$   
Restriction is extended elementwise to t-sets by  $\sigma|_V := \{\sigma'|_V \mid \sigma' \in \sigma\}$ .

**Definition 22.** Define the parallel composition of t-substitutions  $\sigma' \diamond \tau'$  by  
 $\mathbf{cr}(\sigma'_1, \dots, \sigma'_n) \diamond \mathbf{cr}'(\tau'_1, \dots, \tau'_m) :=$   

$$\begin{cases} (\mathbf{cr} \diamond \mathbf{cr}') [(\sigma'_1 \diamond \tau'_1) \times \dots \times (\sigma'_n \diamond \tau'_n) \times \{\tau'_{n+1}\} \times \dots \times \{\tau'_m\}] & \text{if } \mathbf{cr} \diamond \mathbf{cr}' \text{ is defined, and } n \leq m \\ (\mathbf{cr} \diamond \mathbf{cr}') [(\sigma'_1 \diamond \tau'_1) \times \dots \times (\sigma'_m \diamond \tau'_m) \times \{\sigma'_{m+1}\} \times \dots \times \{\sigma'_n\}] & \text{if } \mathbf{cr} \diamond \mathbf{cr}' \text{ is defined, and } m \leq n \\ \{\} & \text{if } \mathbf{cr} \diamond \mathbf{cr}' \text{ is undefined} \end{cases}$$
  
 $\sigma' \diamond \tau'$  yields a set with at most one t-substitution. Parallel composition is extended elementwise to t-sets by  $\sigma \diamond \tau := \bigcup_{\sigma' \in \sigma, \tau' \in \tau} \sigma' \diamond \tau'$ . Note that  $\sigma' \diamond \tau' = \{\}$  if  $\sigma'$  and  $\tau'$  do not agree on  $dom(\sigma') \cap dom(\tau')$ .

**Definition 23.** Define the lifting of a ground constructor term  $u$  to a t-substitution  $[x := u]$ , using the notation from Def. 13, by  
 $[x := cr] := (x \mapsto cr)$  if  $ar(cr) = 0$   
 $[x := cr(u_1, \dots, u_n)] := (x \mapsto cr) ([x := u_1], \dots, [x := u_n])$  if  $ar(cr) = n > 0$   
Lifting is extended elementwise to sets of ground constructor terms by  $[x := S] := \{[x := u] \mid u \in S\}$ .

$(0_x s_y cons_z(0_y 0_z, nil_z)) (x)$	$= \{0\}$
$(0_x s_y cons_z(0_y 0_z, nil_z)) (y)$	$= \{s(0)\}$
$(0_x s_y cons_z(0_y 0_z, nil_z)) (z)$	$= \{cons(0, nil)\}$
$(0_x s_y cons_z(0_y 0_z, nil_z)) (cons(x, cons(y, nil)))$	$= \{cons(0, cons(s(0), nil))\}$
$(0_x s_y cons_z(0_y 0_z, nil_z)) (x')$	$= \{\}$
$(0_x s_y cons_z(0_y 0_z, nil_z)) _{\{x\}}$	$= 0_x$
$(0_x s_y cons_z(0_y 0_z, nil_z)) _{\{y\}}$	$= s_y(0_y)$
$(0_x s_y cons_z(0_y 0_z, nil_z)) _{\{z\}}$	$= cons_z(0_z, nil_z)$
$(0_x s_y cons_z(0_y 0_z, nil_z)) _{\{x, y\}}$	$= 0_x s_y(0_y)$
$(0_x s_y cons_z(0_y 0_z, nil_z)) _{\{x, z\}}$	$= 0_x cons_z(0_z, nil_z)$
$(0_x s_y cons_z(0_y 0_z, nil_z)) _{\{y, z\}}$	$= s_y cons_z(0_y 0_z, nil_z)$
$(0_x) \diamond (s_y(0_y))$	$= \{0_x s_y(0_y)\}$
$(0_x s_y(0_y)) \diamond (0_x cons_z(0_z, nil_z))$	$= \{0_x s_y cons_z(0_y 0_z, nil_z)\}$
$(0_x cons_z(0_z, nil_z)) \diamond (s_y cons_z(0_y 0_z, nil_z))$	$= \{0_x s_y cons_z(0_y 0_z, nil_z)\}$
$(0_x s_y(0_y)) \diamond (s_y cons_z(0_y 0_z, nil_z))$	$= \{0_x s_y cons_z(0_y 0_z, nil_z)\}$
$(0_x s_y(0_y)) \diamond (0_y cons_z(0_z, nil_z))$	$= \{\}$
$[x := 0]$	$= 0_x$
$[y := s(0)]$	$= s_y(0_y)$
$[z := cons(0, nil)]$	$= cons_z(0_z, nil_z)$

**Fig. 6.** Some example computations according to Defs. 18 – 23

**Definition 24.** Let  $\beta$  be an ordinary idempotent substitution with  $n \geq 1$ , let  $\sigma'$  be a t-substitution with  $ran(\beta) \subset dom(\sigma')$  and  $dom(\beta) \cap dom(\sigma') = \{\}$ , define  $\sigma' \circ \beta := \diamond_{x \in dom(\beta)} [x := \sigma' \beta x]$ . We always have  $dom(\sigma' \circ \beta) = dom(\beta)$ ,  $(\sigma' \circ \beta)v = \sigma'(\beta v)$  for all  $v$  with  $vars(v) \subset dom(\beta)$ , and  $(\sigma' \circ \beta)/\beta = \sigma'$ .



For a t-set  $\sigma$  with the same domain as  $\sigma'$ , define  $\sigma \circ \beta := \bigcup_{\sigma' \in \sigma} \sigma' \circ \beta$ . We have  $\text{dom}(\sigma \circ \beta) = \text{dom}(\beta)$ ,  $(\sigma \circ \beta)v = \sigma(\beta v)$  for all  $v$  with  $\text{vars}(v) \subset \text{dom}(\beta)$ , and  $(\sigma \circ \beta)/\beta = \sigma$ .

**Lemma 25.** Some properties of application, restriction, parallel composition, and abstraction are:

- $\sigma'|_V u = \sigma' u$  if  $\text{vars}(u) \subset V$ ;  $\sigma'|_V u = \{\}$ , else
- $\text{dom}(\sigma'|_V) = \text{dom}(\sigma') \cap V$
- $(\sigma'|_{V_1})|_{V_2} = \sigma'|_{V_1 \cap V_2}$
- $\sigma \subset \tau \Rightarrow \sigma|_V \subset \tau|_V$
- $\sigma|_V u = \sigma u$  if  $\text{vars}(u) \subset V$
- $(\sigma \cap \tau)|_V = \sigma|_V \cap \tau|_V$
- $\diamond$  is associative
- $\sigma \diamond \tau = (\sigma \diamond \tau|_{T \setminus S}) \cap (\sigma|_{S \setminus T} \diamond \tau)$  where  $S = \text{dom}(\sigma)$ ,  $T = \text{dom}(\tau)$
- $\sigma' u = \tau' u \Leftrightarrow \sigma'|_{\text{vars}(u)} = \tau'|_{\text{vars}(u)}$
- $\sigma' \langle x_1, \dots, x_n \rangle = \langle \sigma' x_1, \dots, \sigma' x_n \rangle$
- $\sigma \langle x_1, \dots, x_n \rangle \subset \langle \sigma x_1, \dots, \sigma x_n \rangle$ .

**Definition 26.** Define the factorization  $\sigma'/\beta$  of a t-substitution  $\sigma'$  wrt. to an ordinary substitution  $\beta$  with  $\text{dom}(\beta) \subset \text{dom}(\sigma')$  as follows, let  $k := \text{ar}(\text{cr}_x)$ :

1.  $\sigma'/[x_1:=u_1, \dots, x_n:=u_n] := \sigma'/[x_1:=u_1] \diamond \dots \diamond \sigma'/[x_n:=u_n]$  if  $n > 1$
2.  $\text{cr}(\sigma'_1, \dots, \sigma'_n)/[x:=\text{cr}_x(u_1, \dots, u_k)] := \sigma'_1/[x:=u_1] \diamond \dots \diamond \sigma'_k/[x:=u_k]$  if  $k > 0$
3.  $\text{cr}(\sigma'_1, \dots, \sigma'_n)/[x:=\text{cr}_x] := \{\varepsilon\}$  if  $k = 0$
4.  $\text{cr}(\sigma'_1, \dots, \sigma'_n)/[x:=\text{cr}'(u_1, \dots, u_k)] := \{\}$  if  $\text{cr}_x \neq \text{cr}'$
5.  $\sigma'/[x:=y] := [y:=\sigma'x]$  if  $x \neq y \in \mathcal{V}$

$\sigma'/\beta$  yields a set with at most one t-substitution; it is extended elementwise to t-sets by  $\sigma/\beta := \bigcup_{\sigma' \in \sigma} \sigma'/\beta$ . Note that  $[y:=\sigma'x]$  is a singleton or empty set by Defs. 18 and 23. Factorization by the identity substitution is undefined. We have  $\text{dom}(\sigma'/\beta) = \text{ran}(\beta)$  if  $\sigma'/\beta \neq \{\}$ .

**Lemma 27.** (Pattern-Matching Properties)

- a.  $\sigma'/\beta u = \sigma' u$ , if  $\sigma'/\beta \neq \{\}$
- b.  $\sigma/\beta u = \sigma u \cap \top \beta u$

*Proof.*

- a. Induction on  $n = \# \text{dom}(\beta)$ : show  $n = 1$  by induction on  $u$ , show  $n \rightsquigarrow n + 1$  by induction on  $u$ .
- b. follows from a.

**Example 28.** We have

$$\begin{aligned}
& (0_x s_y(0_y))/[y=s(z)] \\
&= 0_y/[y=z] && \text{by Def. 26.2} \\
&= [z:=(0_y)(y)] && \text{by Def. 26.5} \\
&= [z:=\{0\}] && \text{by Def. 18} \\
&= \{0_z\} && \text{by Def. 23}
\end{aligned}$$

and  $\{0_z\}([y:=s(z)](y)) = \{0_z\}(s(z)) = \{s(0)\} = (0_x s_y(0_y))(y)$  by Def. 18.

but

$$\begin{aligned}
& (0_x s_y(0_y))/[y=s(s(z))] \\
&= 0_y/[y=s(s(z))] && \text{by Def. 26.2} \\
&= \{\} && \text{by Def. 26.4}
\end{aligned}$$

**Lemma 29.** The following propositions are equivalent:

- a.  $\sigma'/\beta \neq \{\}$
- b.  $\sigma'u \cap \top\beta u \neq \{\}$  for all  $u$  with  $\text{vars}(u) \subset \text{dom}(\beta)$
- c.  $\sigma'u \cap \top\beta u \neq \{\}$  for some  $u$  with  $\text{vars}(u) = \text{dom}(\beta)$

*Proof.*

a.  $\Rightarrow$  b. by induction on  $u$ ;

b.  $\Rightarrow$  c. trivial;

c.  $\Rightarrow$  a. Show  $\sigma'u = \tau'\beta u \Rightarrow \sigma'x = \tau'\beta x$  for all  $\tau' \in \top$  and  $x \in \text{vars}(u)$  by induction on  $u$ .

Show  $\sigma'u \cap \tau'\beta u \neq \{\} \Rightarrow \sigma'_{/[x_i=u_i]}y = \sigma'_{/[x_j=u_j]}y$  for all  $\tau' \in \top$  and  $y \in \text{vars}(u_i) \cap \text{vars}(u_j)$ ,  $x_i, x_j \in \text{vars}(u)$ .

Show by induction on  $\#\text{dom}(\beta)$  that both conditions together imply a.

**Lemma 30.** Let  $u_1, \dots, u_n$  have pairwise disjoint variables, and let  $\text{vars}(u_i) \subset \text{dom}(\sigma'_i)$ . Then,  $\sigma'_1 u_1 = \dots = \sigma'_n u_n$  iff  $u_1, \dots, u_n$  are simultaneously unifiable by  $\beta_1 \odot \dots \odot \beta_n$  with  $\text{dom}(\beta_i) = \text{vars}(u_i)$  and  $\sigma'_1/\beta_1 = \dots = \sigma'_n/\beta_n \neq \{\}$ .

*Proof.*

“ $\Rightarrow$ ”: Unifiability is obvious, minimality of  $\beta_i$  implies  $\sigma'_i u_i \cap \top\beta_i u_i \neq \{\}$ , hence  $\sigma'_i/\beta_i \neq \{\}$  by 29.

According to 27,  $\sigma'_i/\beta_i \beta_i u_i = \sigma'_i u_i = \sigma'_j u_j = \sigma'_j/\beta_j \beta_j u_j = \sigma'_j/\beta_j \beta_i u_i$ , hence  $\sigma'_i/\beta_i = \sigma'_j/\beta_j$ .

“ $\Leftarrow$ ”: According to 27, we have  $\sigma'_i u_i = \sigma'_i/\beta_i \beta_i u_i = \sigma'_j/\beta_j \beta_j u_j = \sigma'_j u_j$ .

Domain conditions as in Lemma 30 can always be satisfied by bounded renaming, factorizing by a renaming substitution, cf. Alg. 47 below. If  $u_1$  and  $u_2$  cannot be unified,  $\sigma_1 u_1$  and  $\sigma_2 u_2$  are always disjoint.

**Theorem 31.** Let  $\beta = \text{mgu}(u_1, u_2)$ ,  $\text{dom}(\beta) = \text{vars}(u_1, u_2)$ ,  $\text{vars}(u_i) \subset \text{dom}(\sigma_i)$ ,  $\text{dom}(\sigma_1) \cap \text{dom}(\sigma_2) = \{\}$ , and  $u = \beta u_1$ ; then  $\sigma_1 u_1 \cap \sigma_2 u_2 = (\sigma_1 \diamond \sigma_2)/\beta u$ .

*Proof.*<sup>5</sup>

“ $\subset$ ”: Suppose  $\sigma'_1 u_1 = \sigma'_2 u_2 \subset \sigma_1 u_1 \cap \sigma_2 u_2$ , let  $\gamma$  be a renaming substitution;

then,  $(\sigma'_1 \diamond \sigma'_2)\langle u_1, u_2 \rangle = (\sigma'_1 \diamond \sigma'_2)\langle u_2, u_1 \rangle \stackrel{*}{=} (\sigma'_1 \diamond \sigma'_2)/\gamma \gamma \langle u_2, u_1 \rangle$ .

Applying Lemma 30 yields  $(\sigma'_1 \diamond \sigma'_2)/\beta \neq \{\}$ , since  $\beta \circ \gamma^{-1} = \text{mgu}(\langle u_1, u_2 \rangle, \langle u_2, u_1 \rangle)$ ,

hence  $\sigma'_1 u_1 = (\sigma'_1 \diamond \sigma'_2) u_1 \stackrel{*}{=} (\sigma'_1 \diamond \sigma'_2)/\beta \beta u_1 = (\sigma'_1 \diamond \sigma'_2)/\beta u \subset (\sigma_1 \diamond \sigma_2)/\beta u$ ;

similarly,  $\sigma'_2 u_2 \subset (\sigma_1 \diamond \sigma_2)/\beta u$ .

“ $\supset$ ”: Suppose  $(\sigma'_1 \diamond \sigma'_2)/\beta \subset (\sigma_1 \diamond \sigma_2)/\beta$ , i.e.  $(\sigma'_1 \diamond \sigma'_2)/\beta \neq \{\}$  hence  $\sigma'_1/\beta \neq \{\}$ ,

and  $(\sigma'_1 \diamond \sigma'_2)/\beta u = \sigma'_1/\beta \beta u_1 \stackrel{*}{=} \sigma'_1 u_1 \subset \sigma_1 u_1$ ;

similarly,  $(\sigma_1 \diamond \sigma_2)/\beta u \subset \sigma_2 u_2$ .

The equations marked “ $\stackrel{*}{=}$ ” hold by Lemma 27.

**Theorem 32.**  $\sigma u \neq \{\}$  iff  $\sigma|_{\text{vars}(u)} \cap \mathcal{T}_{(\text{vars}(u) \rightarrow \mathcal{CR})}^* \neq \{\}$ .

Note that  $\mathcal{T}_{(\{x_1, \dots, x_n\} \rightarrow \mathcal{CR})}^* = \text{compose}(\{\text{abstract}(x_1, \mathcal{T}_{\mathcal{CR}}), \dots, \text{abstract}(x_n, \mathcal{T}_{\mathcal{CR}})\})$  is regular since  $\mathcal{T}_{\mathcal{CR}}$  is regular.

<sup>5</sup> Remember that e.g.  $\sigma'_1 u_1$  yields a *set* of ground constructor terms with at most one element.

**Theorem 33.** Let  $u, u_1, \dots, u_n$  have pairwise disjoint variables, let  $\beta_i \circ \gamma_i = mgu(u, u_i)$  exist for all  $i$ . Then,  $\sigma u \subset \tau_1 u_1 \cup \dots \cup \tau_n u_n$  iff  $\forall i \ (\sigma/\beta_i \setminus \tau_i/\gamma_i) \beta_i u \subset \bigcup_{j=1, j \neq i}^n \tau_j u_j$  and  $\sigma \subset \bigcup_{i=1}^n \top \circ \beta_i$ . Note that we provide no algorithm to decide the latter condition.

*Proof.* “ $\Rightarrow$ ”:

Let  $\sigma' \in \sigma$  such that  $\sigma'/\beta_i \neq \{\}$  and  $\sigma'/\beta_i \not\subset \tau_i/\gamma_i$  for all  $i$ .

By assumption,  $j \in \{1, \dots, n\}$  and  $\tau'_j \in \tau_j$  exist such that  $\sigma' u = \tau'_j u_j$ .

By 30,  $\sigma'/\beta_j = \tau'_j/\gamma_j \neq \{\}$ , hence  $j \neq i$ , and  $\sigma'/\beta_i \beta_i u = \sigma' u = \tau'_j u_j$  by 27.

Next, consider an arbitrary  $\sigma' \in \sigma$ . By assumption,  $i$  and  $\tau'_i \in \tau_i$  exist such that  $\sigma' u = \tau'_i u_i$ .

By 30 and 27,  $\sigma'/\beta_i \beta_i u = \sigma' u$ ; hence,  $\{\sigma'\} = \sigma'/\beta_i \circ \beta_i \subset \top \circ \beta_i$

“ $\Leftarrow$ ”:

Let  $\sigma' \in \sigma$ ; by assumption, an  $i$  exists such that  $\sigma' = \tau' \circ \beta_i$  for some  $\tau' \in \top$ .

By 30,  $\sigma'/\beta_i \neq \{\}$ . Case distinction:

- $\sigma'/\beta_i \not\subset \tau_i/\gamma_i$ , then by assumption  $\sigma' u = \sigma'/\beta_i \beta_i u = \tau'_j u_j$  for some  $j \neq i$  and  $\tau'_j \in \tau_j$ .
- $\sigma'/\beta_i = \tau'_i/\gamma_i \neq \{\}$  for some  $\tau'_i \in \tau_i$ , then  $\sigma' u = \sigma'/\beta_i \beta_i u = \tau'_i/\gamma_i \gamma_i u_i = \tau'_i u_i$ .

**Theorem 34.** Let  $u, u_1, \dots, u_n$  have pairwise disjoint variables, let  $u$  be unifiable with each  $u_i$ .

For  $I \subset \{1, \dots, n\}$  let  $\beta_I \circ \bigodot_{i \in I} \beta_{I,i} = mgu(\{u\} \cup \{u_i \mid i \in I\})$  if it exists,

$dom(\beta_I) = vars(u)$ ,  $dom(\beta_{I,i}) = vars(u_i)$  <sup>6</sup>,

let  $J$  be the set of all  $I$  with existing mgu.

Let  $\sigma_I := \{\sigma' \in \sigma \mid \sigma'/\beta_{\{i\}} \neq \{\} \leftrightarrow i \in I\}$ .

Then,  $\sigma u \subset \tau_1 u_1 \cup \dots \cup \tau_n u_n$  iff  $\sigma_I/\beta_I \subset \bigcup_{i \in I} \tau_i/\beta_{I,i}$  for all  $I \in J$ . The latter condition reads  $\sigma_{\{\}} \subset \{\}$  for  $I = \{\}$ . Note that the  $\sigma_I$  are not regular, in general.

*Proof.* First observe (\*): for  $\sigma' \in \sigma_I$ ,  $\{u\} \cup \{u_i \mid i \in I\}$  is simultaneously unifiable since  $\sigma'/\beta_{\{i\}} \beta_{\{i\}} u = \sigma' u = \sigma'/\beta_{\{i\}} \beta_{\{i\},i} u_i$  for all  $i \in I$ .

“ $\Rightarrow$ ”:

Let  $\sigma' \in \sigma_I$  for some  $I \in J$ ; by assumption,  $\sigma' u = \tau'_i u_i$  for some  $i$  and  $\tau'_i \in \tau_i$ .

Applying 30 to  $\sigma' u = \tau'_i u_i$  yields  $\sigma'/\beta_{\{i\}} \neq \{\}$ ; hence  $i \in I$ .

Applying 30 to (\*) and  $\sigma' u = \tau'_i u_i$  yields  $\sigma'/\beta_I = \tau'_i/\beta_{I,i} \neq \{\}$ .

“ $\Leftarrow$ ”:

Let  $\sigma' \in \sigma$ , and let  $I := \{i \mid \sigma'/\beta_{\{i\}} \neq \{\}\}$ . Then,  $\sigma' \in \sigma_I$ , and  $I \in J$  by (\*).

By 35.1 below, it follows that  $\sigma'/\beta_I \neq \{\}$ , hence  $\sigma'/\beta_I = \tau'_i/\beta_{I,i}$  for some  $i \in I$  and  $\tau'_i \in \tau_i$  by assumption.

Hence,  $\sigma' u = \sigma'/\beta_I \beta_I u = \tau'_i/\beta_{I,i} \beta_{I,i} u_i = \tau'_i u_i$ .

**Lemma 35.** Using the notions of 34,

let  $I, I_1, I_2, I_3 \in J$  with  $I_1 \subset I_2 \cap I_3$  and  $I_2 \neq I_3$ ,  $\sigma' \in \sigma$ ,  $\sigma'_2 \in \sigma_{I_2}$ ,  $\sigma'_3 \in \sigma_{I_3}$ , then:

1.  $\sigma'/\beta_I \neq \{\}$  iff  $\forall i \in I \ \sigma'/\beta_{\{i\}} \neq \{\}$ ;
2.  $\sigma'_2/\beta_{I_1} \neq \sigma'_3/\beta_{I_1}$ ;
3.  $\sigma_{I_2/\beta_{I_1}} = \sigma/\beta_{I_1} \setminus \bigcup_{I_3 \in J, I_2 \neq I_3 \supset I_1} \sigma_{I_3/\beta_{I_1}}$ .

*Proof.*

1. “ $\Rightarrow$ ”: Let  $i \in I$ , then  $\sigma' u = \sigma'/\beta_I \beta_I u = \sigma'/\beta_I \beta_{I,i} u_i$ ; by 30,  $\sigma'/\beta_{\{i\}} \neq \{\}$ .

<sup>6</sup> I.e.  $\beta_{\{\}}$  is a renaming substitution on  $u$

1. “ $\Leftarrow$ ”: Let  $I = \{i_1, \dots, i_m\}$ , then  
 $\sigma' u = \sigma' /_{\beta_{\{i_1\}}} \beta_{\{i_1\}} u = \sigma' /_{\beta_{\{i_1\}}} \beta_{\{i_1\}, i_1} u_{i_1} = \dots = \sigma' /_{\beta_{\{i_m\}}} \beta_{\{i_m\}} u = \sigma' /_{\beta_{\{i_m\}}} \beta_{\{i_m\}, i_m} u_{i_m}$ ,  
hence  $\sigma' /_{\beta_I} \neq \{\}$  by 30.
- 2.: By 1., we have  $\sigma'_2 /_{\beta_{I_1}} \neq \{\} \neq \sigma'_3 /_{\beta_{I_1}}$ ; assume  $\sigma'_2 /_{\beta_{I_1}} = \sigma'_3 /_{\beta_{I_1}}$ .  
W.l.o.g., let  $i \in I_3 \setminus I_2$ ,  
then  $\sigma'_2 u = \sigma'_2 /_{\beta_{I_1}} \beta_{I_1} u = \sigma'_3 /_{\beta_{I_1}} \beta_{I_1} u = \sigma'_3 u = \sigma'_3 /_{\beta_{\{i\}}} \beta_{\{i\}} u = \sigma'_3 /_{\beta_{\{i\}}} \beta_{\{i\}, i} u_i$ ,  
hence  $\sigma'_2 /_{\beta_{\{i\}}} \neq \{\}$  contradicting  $i \notin I_2$ .
3. “ $\subset$ ”: Let  $\sigma' \in \sigma_{I_2}$ , then  $\sigma' \in \sigma$  and  $\sigma' /_{\beta_{I_1}} \neq \sigma'' /_{\beta_{I_1}}$  for all  $\sigma'' \in \sigma_{I_3}$ ,  $I_2 \neq I_3 \supset I_1$  by 2.
3. “ $\supset$ ”: Let  $\sigma' \in \sigma$  with  $\sigma' /_{\beta_{I_1}} \neq \{\}$ ; define  $I := \{i \in \{1, \dots, n\} \mid \sigma' /_{\beta_{\{i\}}} \neq \{\}\}$ ,  
then  $I \in J$ , since  $\sigma' \diamond \bigwedge_{i \in I} \sigma' /_{\beta_{\{i\}}} \beta_{\{i\}}$  unifies  $\{u\} \cup \{u_i \mid i \in I\}$ ,  
and  $I_1 \subset I$ , since  $\sigma' /_{\beta_{\{i\}}} \neq \{\}$  for all  $i \in I_1$  by 1.  
Case distinction:
  - (a)  $I \neq I_2$ ; then  $\sigma' /_{\beta_{I_1}} \subset \sigma_{I / \beta_{I_1}}$ , with  $I_2 \neq I \supset I_1$ , hence  $I$  is one of the  $I_3$ ,  
i.e.,  $\sigma' /_{\beta_{I_1}}$  is not contained in the right hand side, and we have nothing to show.
  - (b)  $I = I_2$ ; then,  $\sigma' /_{\beta_{I_1}}$  is contained in the left hand side.

**Example 36.** Let  $J = \{\{\}, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}\}$ , then 35.3 yields following equations, where e.g.  $\sigma_{\{i,j\}}$  is written as  $\sigma_{ij}$ ; similarly for  $\beta$ :

$$\begin{aligned}
\sigma_{12} /_{\beta_{12}} &= \sigma /_{\beta_{12}} \\
\sigma_{13} /_{\beta_{13}} &= \sigma /_{\beta_{13}} \\
\sigma_{12} /_{\beta_1} &= \sigma /_{\beta_1} \setminus (\sigma_{1/\beta_1} \cup \sigma_{13/\beta_1}) \\
\sigma_{1/\beta_1} &= \sigma /_{\beta_1} \setminus (\sigma_{12/\beta_1} \cup \sigma_{13/\beta_1}) \\
\sigma_{12} /_{\beta_2} &= \sigma /_{\beta_2} \setminus \sigma_{2/\beta_2} \\
\sigma_{2/\beta_2} &= \sigma /_{\beta_2} \setminus \sigma_{12/\beta_2} \\
\sigma_{13} /_{\beta_3} &= \sigma /_{\beta_3} \setminus \sigma_{3/\beta_3} \\
\sigma_{3/\beta_3} &= \sigma /_{\beta_3} \setminus \sigma_{13/\beta_3}
\end{aligned}$$

## 4 Regular T-Sets and Algorithms

In this section, we introduce the notion of a regular t-set and provide algorithms to compute with them. We obtain a decidability result for a class of Horn clauses that is isomorphic to regular t-sets (Cor. 43). We present some simple relations like  $x < y$  that can be expressed by regular t-sets (Figs. 7 and 8), and operations on relations that can be computed (Fig. 9).

Using the result from Sect. 3, we can describe regular sets of ground substitutions as subsets of the initial term algebra  $\mathcal{T}_{(V \rightarrow \mathcal{CR})}^*$ . We will only consider t-sets with a unique domain  $\text{dom}(\sigma') = V_{\sigma'}$  for all  $\sigma' \in \sigma$ ; define  $\text{dom}(\sigma) := V_{\sigma}$ . The empty t-set is again denoted by  $\perp$ ; it will be clear from the context whether  $\perp$  denotes the empty sort or the empty t-set. For each finite  $V$ ,  $\top_V := \mathcal{T}_{(V \rightarrow \mathcal{CR})}^*$  is expressible as a regular set. We write  $\top$  for  $\top_V$  when  $V$  is clear from the context; note that  $\mathcal{T}_{(V \rightarrow \mathcal{CR})}^*$  is not expressible since infinitely many t-substitution constructors exist.

We immediately inherit the mechanisms and algorithms given in Sect. 2, i.e. for intersection, relative complement, and inhabitation. In addition, the operations defined in 18, 21, 22, and 23 can be computed for regular t-sets.

**Algorithm 37.** The following algorithm computes the elementwise application of a regular t-set to a variable. Let  $\sigma$  be the name of a regular t-set, and let  $S$  be a new sort name. Define  $\text{apply}(\sigma, x) = S$ , where the algorithm introduces a new sort definition for  $S$ :

T-sets, described as regular sets:		
$Nat_x$	$\doteq 0_x \mid s_x(Nat_x)$	$\doteq \{[x := s^i(0)] \mid i \in \mathbb{N}\}$
$Nat_y$	$\doteq 0_y \mid s_y(Nat_y)$	$\doteq \{[y := s^i(0)] \mid i \in \mathbb{N}\}$
$Nat_{x=y}$	$\doteq 0_x 0_y \mid s_x s_y(Nat_{x=y})$	$\doteq \{[x := s^i(0), y := s^i(0)] \mid i \in \mathbb{N}\}$
$Nat_{x,y}$	$\doteq 0_x 0_y \mid s_x s_y(Nat_{x,y}) \mid$ $0_x s_y(Nat_y) \mid s_x 0_y(Nat_x)$	$\doteq \{[x := s^i(0), y := s^j(0)] \mid i, j \in \mathbb{N}\}$
$Nat_{x < y}$	$\doteq 0_x s_y(Nat_y) \mid s_x s_y(Nat_{x < y})$	$\doteq \{[x := s^i(0), y := s^j(0)] \mid i, j \in \mathbb{N}, i < j\}$

**Fig. 7.** Examples of regular t-sets

1. If  $apply(\sigma, x)$  has already been called earlier,  $S$  is already defined (loop check).
2. Else, if  $\sigma \doteq \sigma_1 \mid \dots \mid \sigma_n$ , define  $S \doteq apply(\sigma_1, x) \mid \dots \mid apply(\sigma_n, x)$
3. Else, if  $\sigma \doteq \mathbf{cr}(\sigma_1, \dots, \sigma_n)$  with  $x \in \text{dom}(\mathbf{cr})$ ,  
define  $S \doteq \mathbf{cr}_x(apply(\sigma_1, x), \dots, apply(\sigma_{ar(\mathbf{cr}_x)}, x))$ ,
4. Else, define  $S \doteq \perp$ .

Using the t-set version of Thm. 7 with  $p_S(u) :\Leftrightarrow u \in \sigma^M x$  if  $S = apply(\sigma, x)$ , it can be shown that  $apply(\sigma, x)^M = \sigma^M x$ . The algorithm needs at most  $\#use(\sigma)$  recursive calls to compute  $apply(\sigma, x)$ .

Although t-substitutions are homomorphic wrt. all constructors in  $\mathcal{CR}$ , t-sets are generally not; e.g., using the definitions from Fig. 7,  $Nat_{x < y}^M(\langle x, y \rangle) \subsetneq \langle Nat_{x < y}^M(x), Nat_{x < y}^M(y) \rangle$ , cf. Lemma 25. Such t-sets can express certain relations between distinct variables, e.g.  $Nat_{x < y}$  always assigns a value to  $x$  that is less than the value assigned to  $y$ . Figure 8 shows some more nontrivial relations that are expressible by regular t-sets. Figure 9 shows operations on relations that can be computed for t-sets.

**Definition 38.** We call a t-set  $\sigma$  independent if it is homomorphic on linear terms, i.e. if  $\sigma\langle x_1, \dots, x_n \rangle = \langle \sigma x_1, \dots, \sigma x_n \rangle$ , otherwise we call it “dependent”. An independent t-set assigns the value of one variable independently of the value of the others, e.g.  $Nat_{x,y}$  in Fig. 7. A finite union  $\sigma_1 \cup \dots \cup \sigma_n$  of independent t-sets  $\sigma_i$  is called semi-independent. The intersection of two (semi-)independent t-sets is again (semi-)independent; the union of two semi-independent t-sets is trivially semi-independent.

**Algorithm 39.** The following algorithm computes the elementwise application of a regular t-set to a linear constructor term. Let  $u$  be a linear constructor term, let  $\sigma$  be the name of a regular t-set such that  $\sigma|_{\text{vars}(u)}$  is independent. Define  $apply(\sigma, cr(u_1, \dots, u_n)) \doteq cr(apply(\sigma, u_1), \dots, apply(\sigma, u_n))$ ; if  $u \in \mathcal{V}$ , compute  $apply(\sigma, u)$  by Alg. 37. Then,  $apply(\sigma, u)^M = \sigma^M u$ .

**Algorithm 40.** The following algorithm computes the elementwise restriction of a regular t-set to a set of variables. Let  $\sigma$  be the name of a regular t-set,  $V \subset \mathcal{V}$ , and let  $\tau$  be a new name for a regular t-set. Define  $restrict(\sigma, V) = \tau$ , where the algorithm introduces a new t-set definition for  $\tau$ :

1. If  $restrict(\sigma, V)$  has already been called earlier,  $\tau$  is already defined (loop check).
2. Else, if  $\sigma \doteq \sigma_1 \mid \dots \mid \sigma_n$ , define  $\tau \doteq restrict(\sigma_1, V) \mid \dots \mid restrict(\sigma_n, V)$
3. Else, if  $\sigma \doteq \mathbf{cr}(\sigma_1, \dots, \sigma_n)$ , define  $\tau \doteq (\mathbf{cr}|_V)(restrict(\sigma_1, V), \dots, restrict(\sigma_m, V))$ ,  
where  $m = ar(\mathbf{cr}|_V)$ .

Using Thm. 7 with  $p_\tau(\tau') :\Leftrightarrow \tau' \in \sigma^M|_V$  if  $\tau = restrict(\sigma, V)$ , it can be shown that  $restrict(\sigma, V)^M = \sigma^M|_V$ . The algorithm needs at most  $\#use(\sigma)$  recursive calls to compute  $restrict(\sigma, V)$ . If  $\sigma$  is (semi-)independent, then so is  $restrict(\sigma, V)$ .

**Algorithm 41.** The following algorithm computes the elementwise parallel composition of two regular t-sets  $\sigma$  and  $\tau$ . Let  $\mu$  be a new name for a regular t-set. Define  $compose(\sigma, \tau) = \mu$ , where the algorithm introduces a new t-set definition for  $\mu$ :

1. If  $compose(\sigma, \tau)$  has already been called earlier,  $\mu$  is already defined (loop check).
2. Else, if  $\sigma \doteq \sigma_1 \mid \dots \mid \sigma_n$ , define  $\mu \doteq compose(\sigma_1, \tau) \mid \dots \mid compose(\sigma_n, \tau)$ .
3. Else, if  $\tau \doteq \tau_1 \mid \dots \mid \tau_n$ , define  $\mu \doteq compose(\sigma, \tau_1) \mid \dots \mid compose(\sigma, \tau_n)$ .
4. Else, if  $\sigma \doteq \mathbf{cr}(\sigma_1, \dots, \sigma_n)$ ,  $\tau \doteq \mathbf{cr}'(\tau_1, \dots, \tau_m)$ ,  $\mathbf{cr}$  and  $\mathbf{cr}'$  agree on their domain intersection, and w.l.o.g.  $n \leq m$ , define  $\mu \doteq (\mathbf{cr} \diamond \mathbf{cr}')(compose(\sigma_1, \tau_1), \dots, compose(\sigma_n, \tau_n), \tau_{n+1}, \dots, \tau_m)$ .
5. Else, if  $\sigma \doteq \mathbf{cr}(\sigma_1, \dots, \sigma_n)$ ,  $\tau \doteq \mathbf{cr}'(\tau_1, \dots, \tau_m)$ , and  $\mathbf{cr}$  and  $\mathbf{cr}'$  do not agree on their domain intersection, define  $\mu \doteq \perp$ .

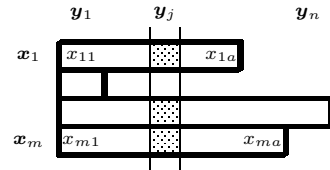
Using Thm. 7 with  $p_\mu(\mu') :\Leftrightarrow \mu' \in \sigma^M \diamond \tau^M$  if  $\mu = compose(\sigma, \tau)$ , it can be shown that  $compose(\sigma, \tau)^M = \sigma^M \diamond \tau^M$ . The algorithm needs at most  $\#use(\sigma) * \#use(\tau)$  recursive calls to compute  $compose(\sigma, \tau)$ . If  $\sigma$  and  $\tau$  are both (semi-)independent, then so is  $compose(\sigma, \tau)$ . We write  $compose(\{\sigma_1, \dots, \sigma_n\})$  for  $compose(\sigma_1, compose(\dots, compose(\sigma_{n-1}, \sigma_n) \dots))$ .

**Algorithm 42.** The following algorithm computes the elementwise lifting of a regular sort to a regular t-set. Let  $S$  be the name of a regular sort,  $x \in \mathcal{V}$ , and let  $\sigma$  be a new name for a regular t-set. Define  $abstract(S, x) = \sigma$ , where the algorithm introduces a new t-set definition for  $\sigma$ :

1. If  $abstract(S, x)$  has already been called earlier,  $\sigma$  is already defined (loop check).
2. Else, if  $S \doteq S_1 \mid \dots \mid S_n$ , define  $\sigma \doteq abstract(S_1, x) \mid \dots \mid abstract(S_n, x)$ .
3. Else, if  $S \doteq cr(S_1, \dots, S_n)$ , define  $\sigma \doteq (x \mapsto cr)(abstract(S_1, x), \dots, abstract(S_n, x))$ .

Using Thm. 7 with  $p_\sigma(\sigma') :\Leftrightarrow \sigma' \in [x := S^M]$  if  $\sigma = abstract(S, x)$ , it can be shown that  $abstract(S, x)^M = [x := S^M]$ . The algorithm needs at most  $\#use(S)$  recursive calls to compute  $abstract(x, S)$ .  $abstract(x, S)$  always yields an independent t-set.

Regular sorts from Sect. 2 can be shown to correspond to Horn clauses with unary predicates and thus decide this theory class by extending the form of sort expressions allowed on the right-hand side of a sort definition to include intersections, too.



Similarly, regular t-sets correspond to Horn clauses of the following form:

$p(cr_1(\mathbf{x}_1), \dots, cr_m(\mathbf{x}_m)) \leftarrow p_1(\mathbf{y}_1) \wedge \dots \wedge p_n(\mathbf{y}_n)$  where  $\mathbf{x}_i := \langle x_{i1}, \dots, x_{ia_i} \rangle$  for  $i = 1, \dots, m$ , and  $\mathbf{y}_j := \langle x_{ij} \mid i = 1, \dots, m, j \leq ar(cr_i) \rangle$  for  $j = 1, \dots, n$  with  $n := \max_{i=1, \dots, m} ar(cr_i)$ . The relation between  $\mathbf{x}_i$  and  $\mathbf{y}_j$  is shown in the above diagram. If all term constructors  $cr_i$  have the same arity  $n$ , the  $\mathbf{y}_j$  are the column vectors of an  $m \times n$  matrix built from the  $\mathbf{x}_i$  as line vectors. For example, the definition  $Lgth_{x,y} \doteq 0_x nil_y \mid s_x snoc_y(Lgth_{x,y}, Nat_y)$  corresponds to the Horn clauses  $lgth(0, nil)$  and  $lgth(s(x), snoc(y_1, y_2)) \leftarrow lgth(x, y_1) \wedge nat(y_2)$ . We thus have the following

**Corollary 43.** The satisfiability of any predicate defined by Horn clauses of the above form can be decided. The set of such predicates is closed wrt. conjunction, disjunction, and negation.

**Algorithm 44.** The following algorithm “duplicates” each t-substitution  $\sigma'$  in  $\sigma$ , i.e., it composes  $\sigma'$  with a renamed copy of itself. Let  $\sigma$  be the name of a regular t-set, let  $\beta$  be an ordinary idempotent

Expressible relations e.g.:

prefix  $x$  of length  $y$  of a *snoc*-list  $z$  with regular element sort  $Elem$

$$Pref_{x,y,z} \doteq nil_x 0_y nil_z \mid snoc_x s_y snoc_z (Pref_{x,y,z}, Elem_{x=z}) \mid nil_x 0_y snoc_z (List_x, Elem_x)$$

$$List_x \doteq nil_x \mid snoc_x (List_x, Elem_x)$$

$$Elem_x = abstract(x, Elem)$$

$$Elem_{x=z} = dup(Elem_x, [z:=x])$$

lexicographical order on *cons*-lists wrt. regular element ordering  $Elem_{x<y}$

$$Lex_{x<y} \doteq nil_x cons_y (Elem_y, List_y) \mid cons_x cons_y (Elem_{x=y}, Lex_{x<y}) \\ \mid cons_x cons_y (Elem_{x<y}, List_{x,y})$$

$$Elem_x = abstract(x, Elem)$$

$$Elem_y = abstract(y, Elem)$$

$$Elem_{x=y} = dup(Elem_x, [y:=x])$$

$$List_x \doteq nil_x \mid cons_x (Elem_x, List_x)$$

$$List_y \doteq nil_y \mid cons_y (Elem_y, List_y)$$

$$List_{x,y} = compose(List_x, List_y)$$

matching of tree  $x$  at the root of tree  $y$  (variable bindings not considered)

set of functions symbols  $F$ , only one variable symbol  $v$

$$Mtc_{x,y} \doteq \bigvee_{f \in F} f_x f_y (Mtc_{x,y}, \dots, Mtc_{x,y}) \mid v_x f_y (Term_y, \dots, Term_y)$$

$$Term \doteq v \mid \bigvee_{f \in F} f(Term, \dots, Term)$$

$$Term_y = abstract(y, Term)$$

sum  $z$  of binary strings  $x$  and  $y$  (*cons*-lists, least bit first)

$$Sum_{x,y,z} \doteq Sum_{0,x,y,z}$$

$$Sum_{0,x,y,z} \doteq nil_x nil_y nil_z \\ \mid cons_x nil_y cons_z (i_x i_z \mid o_x o_z, Bin_{0,x=z}) \\ \mid cons_x cons_y cons_z (o_x o_y o_z \mid o_x i_y i_z \mid i_x o_y i_z, Sum_{0,x,y,z}) \\ \mid cons_x cons_y cons_z (i_x i_y o_z, Sum_{1,x,y,z})$$

$$Sum_{1,x,y,z} \doteq nil_x nil_y cons_z (i_z, nil_z) \\ \mid cons_x nil_y cons_z (i_x o_z, Bin_{1,x=z}) \\ \mid cons_x cons_y cons_z (o_x o_y i_z, Sum_{0,x,y,z}) \\ \mid cons_x cons_y cons_z (o_x i_y o_z \mid i_x o_y o_z \mid i_x i_y i_z, Sum_{1,x,y,z})$$

$$Bin' \doteq nil \mid cons(o, Bin') \mid cons(i, Bin')$$

$$Bin_x = abstract(x, Bin')$$

$$Bin_{0,x=z} = dup(Bin_x, [z:=x])$$

$$Bin_{1,x=z} \doteq nil_x cons_z (i_z, nil_z) \mid cons_x cons_z (o_x i_z, Bin_{0,x=z}) \\ \mid cons_x cons_z (i_x o_z, Bin_{1,x=z})$$

**Fig. 8.** Some relations expressible by regular t-sets

Operations on relations e.g.:	
relation join	$\sigma \diamond \tau$
relational image	
$R[x_0] = \{y \mid x_0 R y\}$	$R[x_0] = \text{apply}(\text{compose}(R, \text{abstract}(x, x_0)), y)$
factorization wrt. equivalence	
$x R_E y \Leftrightarrow \exists x', y' \ x E x' R y' E y$	$R_E = \text{compose}(\{E, R, E\})$
equivalence from mapping	
$x E_M y \Leftrightarrow M(x) = M(y)$	$E_M = \text{compose}(\text{fact}(M, [x := x', y := y']), \text{fact}(M, [x := y', y := x']))$
restriction	$\sigma _V$
bounded renaming	$\sigma/\beta$
conjunction	$\sigma \cap \tau$
disjunction	$\sigma \mid \tau$
negation	$\top \setminus \sigma$

For example, using the definitions from Fig. 8,  $\text{restrict}(\text{Pref}_{x,y,z}, \{x, y\})$  yields the length function on *snoc*-lists, and  $\text{apply}(\text{compose}(\text{Pref}_{x,y,z}, \text{abstract}(y, s^3(0))), x)$  yields the regular sort of all *snoc*-lists of length 3.

**Fig. 9.** Computable operations on relations in t-set form

substitution with  $\beta x \in \mathcal{V}$  for all  $x \in \text{dom}(\beta)$ ,  $\beta$  need not be linear. Let  $\tau$  be a new name for a regular t-set. Define  $\text{dup}(\sigma, \beta) = \tau$ , where the algorithm introduces a new t-set definition for  $\tau$ :

1. If  $\text{dup}(\sigma, \beta)$  has already been called earlier,  $\tau$  is already defined (loop check).
2. Else, if  $\sigma \doteq \sigma_1 \mid \dots \mid \sigma_n$ , define  $\tau \doteq \text{dup}(\sigma_1, \beta) \mid \dots \mid \text{dup}(\sigma_n, \beta)$ .
3. Else, if  $\sigma \doteq \mathbf{cr}(\sigma_1, \dots, \sigma_n)$  and  $\text{cr}_{x_i} = \text{cr}_{x_j}$  whenever  $\beta x_i = \beta x_j$ , let  $\mathbf{cr}'$  be a t-substitution constructor such that  $\mathbf{cr}'_{\beta x} = \mathbf{cr}_x$  for all  $x \in \text{dom}(\beta)$ , define  $\tau \doteq (\mathbf{cr} \diamond \mathbf{cr}')(\text{dup}(\sigma_1, \beta), \dots, \text{dup}(\sigma_n, \beta))$ .
4. Else, define  $\theta \doteq \perp$ .

Using Thm. 7 with  $p_\tau(\tau') : \Leftrightarrow \tau' \in \bigcup_{\sigma' \in \sigma} \sigma' \diamond (\sigma'/\beta)$ , it can be shown that  $\text{dup}(\sigma, \beta)^M = \bigcup_{\sigma' \in \sigma} \sigma' \diamond (\sigma'/\beta)$ . The algorithm needs at most  $\#use(\sigma)$  recursive calls to compute  $\text{dup}(\sigma, \beta)$ .

**Example 45.** Using the definitions in Fig. 7, we get  $\text{dup}(\text{Nat}_x, [y := x]) = \text{Nat}_{x=y}$ .

**Algorithm 46.** If  $\sigma$  is regular and  $\beta x \in \mathcal{V}$  for all  $x \in \text{dom}(\beta)$ ,  $\sigma \circ \beta$  is again regular; in general, it is not. In the former case, the following algorithm computes a regular t-set definition for  $\sigma \circ \beta$ :

1. If  $\beta = \beta_1 \circ \beta_2$  such that  $\beta_1$  and  $\beta_2$  are each injective, i.e. renamings, let  $\gamma$  be a renaming on  $\text{ran}(\beta_2)$ , then  $\sigma \circ \beta = \text{fact}(\text{dup}(\sigma, \gamma), \beta_1^{-1} \circ (\beta_2^{-1} \circ \gamma^{-1}))$ .
2. Any other  $\beta$  can be represented as  $\beta_1 \circ \dots \circ \beta_n$  such that  $1 \leq \#\{x \mid \beta_i x = y\} \leq 2$  for all  $y$  and for all  $i$ , i.e. each  $\beta_i$  has the form required by 1.; then  $\sigma \circ \beta = (\dots (\sigma \circ \beta_1) \circ \dots) \circ \beta_n$ .

**Algorithm 47.** The following algorithm computes  $\text{fact}(\sigma, \beta)$  if  $\beta x \in \mathcal{V}$  for all  $x$ . Let  $\sigma$  be a regular t-set, let  $\mu$  be a new name for a regular t-set; define  $\text{fact}(\sigma, \beta) = \mu$ , where a new t-set definition is introduced for  $\mu$ :

1. If  $\text{fact}(\sigma, \beta)$  has been called earlier,  $\mu$  is already defined (loop checking).



2. Else, if  $\sigma \doteq \sigma_1 \mid \dots \mid \sigma_n$ , define  $\mu \doteq fact(\sigma_1, \beta) \mid \dots \mid fact(\sigma_n, \beta)$ .
3. Else, if  $\sigma \doteq \mathbf{cr}(\sigma_1, \dots, \sigma_n)$  with  $dom(\beta) \subset dom(\mathbf{cr})$ ,  
and  $\mathbf{cr}_x = \mathbf{cr}_y$  whenever  $\beta x = \beta y$ , define  $\mathbf{cr}' : ran(\beta) \rightarrow \mathcal{CR}$  by  $\mathbf{cr}'_{\beta x} := \mathbf{cr}_x$ ,  
define  $\mu \doteq \mathbf{cr}'(fact(\sigma_1, \beta), \dots, fact(\sigma_{ar(\mathbf{cr}')}(\beta)))$ .
4. Else, define  $\mu \doteq \perp$ .

Using the induction principle from Thm. 7, lifted to t-sets, with  $p_\mu(\sigma') :\Leftrightarrow \sigma' \in \sigma^M/\beta$  if  $\mu = fact(\sigma, \beta)$ , it can be shown that  $fact(\sigma, \beta)^M = \sigma^M/\beta$ . The algorithm needs at most  $\#use(\sigma)$  recursive calls to compute  $fact(\sigma, \beta)$ .

**Definition 48.**  $\beta$  is called homogeneous if all variables in the range of  $\beta$  occur at the same depth, i.e., if  $\beta x \in \mathcal{V}$  for each  $x \in dom(\beta)$ , or if  $\beta x = \mathbf{cr}_x(u_{x1}, \dots, u_{x ar(\mathbf{cr}_x)})$  for all  $x \in dom(\beta)$  and appropriate  $u_{xi}$ , and  $[x := u_{xi} \mid x \in dom(\beta), ar(\mathbf{cr}_x) \geq i]$  is again homogeneous for each  $i$ .

**Algorithm 49.** The following algorithm computes  $fact(\sigma, \beta)$  if  $\beta$  is homogeneous. Let  $\sigma$  be a regular t-set, define  $fact(\sigma, \beta)$  by:

1. If  $\beta = [\ ]$ , define  $fact(\sigma, \beta) := \{\varepsilon\}$ .
2. Else, if  $\beta x \in \mathcal{V}$  for all  $x$ , compute  $fact(\sigma, \beta)$  by Alg. 47.
3. Else, if  $\sigma \doteq \sigma_1 \mid \dots \mid \sigma_n$ , define  $fact(\sigma, \beta) := fact(\sigma_1, \beta) \mid \dots \mid fact(\sigma_n, \beta)$ .
4. Else, if  $\sigma \doteq \mathbf{cr}(\sigma_1, \dots, \sigma_n)$ ,  $dom(\beta) \subset dom(\mathbf{cr})$ , and  $\beta x = \mathbf{cr}_x(u_{x1}, \dots, u_{x ar(\mathbf{cr}_x)})$  for all  $x \in dom(\beta)$ ,  
define  $fact(\sigma, \beta) := compose(\{ fact(\sigma_1, [x := u_{x1} \mid x \in dom(\beta), ar(\mathbf{cr}_x) \geq 1]), \dots,$   
 $fact(\sigma_n, [x := u_{x,n} \mid x \in dom(\beta), ar(\mathbf{cr}_x) \geq n]) \})$ .
5. Else, define  $fact(\sigma, \beta) := \perp$ .

Using the lexicographic combination of the size of range terms of  $\beta$  and  $\dot{<}$ , it can be shown that the algorithm always terminates and yields  $fact(\sigma, \beta)^M = \sigma^M/\beta$  for  $\beta \neq [\ ]$ . The algorithm needs at most  $depth(\beta)$  recursive calls to compute  $fact(\sigma, \beta)$ . If  $\sigma$  is semi-independent, then so is  $fact(\sigma, \beta)$ .

**Algorithm 50.** Let  $\beta$  be pseudolinear,  $dom(\beta) \subset dom(\sigma)$ ,  $V := \{x \in dom(\beta) \mid \beta x \in \mathcal{V}\}$ . Define the finite set  $hom(\sigma, \beta)$  of ordinary homogenizing substitutions for  $\beta$  wrt.  $\sigma$ :

1. If  $\sigma \doteq \sigma_1 \mid \dots \mid \sigma_n$ , define  $hom(\sigma, \beta) := hom(\sigma_1, \beta) \cup \dots \cup hom(\sigma_n, \beta)$ .
2. Else, if  $\sigma \doteq \mathbf{cr}(\sigma_1, \dots, \sigma_n)$ , let  $\gamma_0 := [\beta x := \mathbf{cr}_x(y_{x1}, \dots, y_{x ar(\mathbf{cr}_x)}) \mid x \in V]$  where the  $y_{x,i}$  are new variables, define  $hom(\sigma, \beta) := (hom(\sigma, \gamma_0 \circ \beta) \circ \gamma_0)|_{ran(\beta)}$ .
3. Else, if  $\sigma \doteq \mathbf{cr}(\sigma_1, \dots, \sigma_n)$ ,  $\beta$  not homogeneous,  $V = \{\}$ ,  
and  $\beta x = \mathbf{cr}_x(u_{x1}, \dots, u_{x ar(\mathbf{cr}_x)})$  for all  $x \in dom(\beta)$ ,  
define  $hom(\sigma, \beta) := \bigcirc_{i=1}^n hom(\sigma_i, [x := u_{xi} \mid x \in dom(\beta), ar(\mathbf{cr}_x) \geq i])$ .
4. Else, if  $\beta$  homogeneous, define  $hom(\sigma, \beta) := \{[x := y_x \mid x \in ran(\beta)]\}$ ,  
where each  $y_x$  is a new variable.
5. Else, define  $hom(\sigma, \beta) := \{\}$ .

Then, for each  $\gamma \in hom(\sigma, \beta)$ :

1.  $\gamma \circ \beta$  is homogeneous,
2.  $dom(\gamma) = ran(\beta)$ ,
3. for each  $\sigma' \in \sigma$  with  $\sigma'/\beta \neq \{\}$  there exists a  $\gamma \in hom(\sigma, \beta)$  such that  $\sigma'/\gamma \circ \beta \neq \{\}$ , and

4. for each  $\sigma' \in \sigma$  with  $\sigma'/\beta \neq \{\}$  there exists at most one  $\gamma \in \text{hom}(\sigma, \beta)$  such that  $\sigma'/\gamma \circ \beta \neq \{\}$ .

*Proof.* Use the lexicographic combination of the size of range terms of  $\beta$  and  $\dot{<}$  as termination ordering and as induction ordering for 1. to 4. The algorithm needs at most  $\#use(\sigma) * \text{depth}(\beta)$  recursive calls to compute  $\text{hom}(\sigma, \beta)$ .

**Theorem 51.** If  $\sigma$  regular and  $\beta$  pseudolinear, then  $\sigma/\beta u = \bigcup_{\gamma \in \text{hom}(\sigma, \beta)} \sigma/\gamma \circ \beta \gamma u$  for all  $u$  with  $\text{vars}(u) \subset \text{ran}(\beta)$ , where the  $\sigma/\gamma \circ \beta$  are regular.

*Proof.* Regularity follows from 50 and 49. Since for each  $\{\} \neq \sigma'/\beta \in \sigma/\beta$  there exists exactly one  $\gamma$  with  $\sigma'/\gamma \circ \beta \neq \{\}$  by 50, the claimed equality follows from 34.

**Example 52.** Consider the definition of  $\text{nat}_{x < y}$  in Fig. 7; let  $\beta := [x := x', y := s(y')]$ . We first homogenize  $\beta$  wrt.  $\text{nat}_{x < y}$  by 50, yielding  $\text{hom}(\text{nat}_{x < y}, \beta) = \{[x' := 0, y' := y''], [x' := s(x''), y' := y'']\}$ . Then, using 49, we factorize  $\text{nat}_{x < y}$  wrt. the homogeneous substitutions  $[x' := 0, y' := y''] \circ \beta$  and  $[x' := s(x''), y' := y''] \circ \beta$ , yielding  $\text{fact}(\text{nat}_{x < y}, [x := 0, y := s(y'')]) = \text{nat}_{y''}$  and  $\text{fact}(\text{nat}_{x < y}, [x := s(x''), y := s(y'')]) = \text{nat}_{x'' < y''}$ , respectively. Using 51, we can thus compute  $\text{nat}_{x < y}/\beta \langle x', y' \rangle$  as  $\text{nat}_{y''} \langle 0, y'' \rangle \mid \text{nat}_{x'' < y''} \langle s(x''), y'' \rangle$ .

**Example 53.** Let  $\sigma \doteq 0_x \text{cr}_y(0_y, 0_y) \mid \text{cr}_x \text{cr}_y(\sigma, \sigma)$ , and  $\beta = [x := x', y := \text{cr}(y', x')]$ . Then,  $\sigma^M/\beta$  is the infinite set of complete binary trees  $A$  that is minimal with  $0_x 0_{y'} \in A$  and  $\text{cr}_x \text{cr}_{y'}(\sigma', \sigma') \in A$  for  $\sigma' \in A$ .  $\sigma/\beta x'$  is a similar set of complete binary trees which cannot be written as  $\tau_1 u_1 \cup \dots \cup \tau_n u_n$  with regular  $\tau_i$ .

*Proof.*

1. Show  $\sigma' \in A \Rightarrow \sigma' \in \sigma^M/\beta$  by induction on  $\sigma'$ .
2. Show  $\sigma^M/\beta = \{0_x 0_{y'}\} \cup (\sigma^M/[x := x_1, y := y'] \diamond \sigma^M/[x' := x_2, y' := x_1]) \circ [x' := \text{cr}(x_1, x_2)]$  by direct computation.
3. Show  $\sigma^M/[x := x_1, y := y'] \diamond \sigma'/[x' := x_2, y' := x_1] = \sigma'/[x' := x_2, y' := x_1] \diamond \text{cr}_{y'}(\sigma'|_{y'}, \sigma'|_{y'})$  by induction on  $\sigma'$  using 2.
4. Show  $\sigma^M/\beta \subset A$  by induction on the order  $\sigma'_1 < \sigma'_2 \Leftrightarrow \sigma'_1 x' \not\leq \sigma'_2 x'$  using 3. and 4.
5. Show that no infinite set of complete binary trees can be represented as  $\tau u$  with regular t-set  $\tau$  and constructor term  $u$  by induction on  $u$ , using in the base case 37 and a pumping lemma.

**Theorem 54.** If  $\sigma$  is independent, then  $\sigma/\beta = \diamond_{x \in \text{dom}(\beta)} \sigma/[x := \beta x]$ .

The right-hand side can be algorithmically computed using 49, since  $[x := \beta x]$  is always homogeneous.

**Algorithm 55.** Let  $\sigma$  be the name of a regular t-set. The following algorithm decides whether  $\sigma \subset \top \circ \beta$ ,  
i.e. whether  $\sigma'/\beta \neq \{\}$  for all  $\sigma' \in \sigma$ .

Define  $div(\sigma, \beta) :\Leftrightarrow \bigwedge_{x \in dom(\beta)} div(\sigma, [x := \beta x])$   
 $\wedge \bigwedge_{x, x' \in dom(\beta), x \neq x'} \bigwedge_{y \in vars(\beta x) \cap vars(\beta x')}$   
 $single(apply(fact(\sigma, [x := \beta x]), y) \mid apply(fact(\sigma, [x' := \beta x']), y));$   
 where  $div(\sigma, [x := u])$  is computed as follows:

1. If  $\sigma \doteq \sigma_1 \mid \dots \mid \sigma_n$ , define  $div(\sigma, [x := u]) :\Leftrightarrow div(\sigma_1, [x := u]) \wedge \dots \wedge div(\sigma_n, [x := u])$ .
2. Else, if  $\sigma \doteq \mathbf{cr}(\sigma_1, \dots, \sigma_n)$  and  $u = \mathbf{cr}_x(u_1, \dots, u_k)$ ,  
 define  $div(\sigma, [x := u]) :\Leftrightarrow \bigwedge_{i=1}^k div(\sigma_i, [x := u_i])$   
 $\wedge \bigwedge_{1 \leq i < j \leq k} \bigwedge_{y \in vars(u_i) \cap vars(u_j)}$   
 $single(apply(fact(\sigma_i, [x := u_i]), y) \mid apply(fact(\sigma_j, [x := u_j]), y)).$
3. Else, if  $\sigma \doteq \mathbf{cr}(\sigma_1, \dots, \sigma_n)$  and  $u = \mathbf{cr}'(u_1, \dots, u_k)$  with  $\mathbf{cr}' \neq \mathbf{cr}_x$ , define  $div(\sigma, [x := u]) :\Leftrightarrow false$ .
4. Else, if  $u \in \mathcal{V}$  (also  $u = x$ ), define  $div(\sigma, [x := u]) :\Leftrightarrow x \in dom(\sigma)$ .

Using the lexicographical combination of  $\triangleleft$  and  $\dot{<}$ , the correctness and termination of the computation of  $div(\sigma, [x := u])$  can be shown by induction on  $\langle \sigma, u \rangle$ . The proof, as well as the correctness proof for  $div(\sigma, \beta)$ , uses the fact that  $\sigma y \cup \tau y$  is a singleton set for all  $y \in dom(\sigma) \cap dom(\tau)$  iff  $\sigma' \diamond \tau' \neq \{\}$  for all  $\sigma' \in \sigma, \tau' \in \tau$ . The algorithm needs at most  $\#use(\sigma)$  recursive calls to decide  $div(\sigma, [x := u])$ .

## 5 Extended Sorts

In this section, we discuss several possible ways of defining a class of sorts that can express more subsets of  $\mathcal{T}_{CR}$  than regular tree languages. In particular, we define a class called “extended sorts” that can express for arbitrary  $u \in \mathcal{T}_{CR, \mathcal{V}}$  the set of all possible values  $U$  of  $u$ , mentioned in Sect. 1, as an extended sort. We define the notion of an “annotated term”  ${}^\sigma v$  where the t-set  $\sigma$  indicates the set of admitted ground constructor instances of  $v$ ’s variables, i.e. their sort.

The results obtained in Sect. 4 allow us to define three different language classes which are all proper extensions of regular tree languages. In each class, a language of ground constructor terms is described by applying regular t-sets  $\sigma$  to constructor terms  $u$ , the classes differing in the form that is allowed for  $\sigma$  and  $u$ :

1.  $\sigma u$  with  $\sigma$  semi-independent,  $u$  arbitrary.  
 The intersection can be computed using Thms. 31 and 54. The subset property  $\sigma u \subset \tau_1 u_1$  (hence equivalence and inhabitation) can be decided using Thm. 34 and Lemma 35, since we have  $J = \{\{\}, \{1\}\}$ ,  $\sigma_{\{1\}} \subset \{\} \Leftrightarrow div(\sigma, \beta_{\{1\}})$ , and  $\sigma_{\{1\}/\beta_{\{1\}}} = \sigma/\beta_{\{1\}}$ . However, this class is not closed wrt. union. Any regular sort  $S^M$  from Sect. 2 can be expressed as  $[x := S^M] x$ , but the converse is false, e.g.  $Nat_x^M \langle x, x \rangle$  is not a regular sort, as can be shown using a pumping lemma [4].
2.  $\sigma_1 u_1 \cup \dots \cup \sigma_n u_n$  with  $\sigma_i$  independent,  $u_i$  arbitrary.  
 This is a proper superclass of the class given in 1. The intersection can be computed using Thms. 31 and 54; union is trivial; inhabitation can be decided using Thm. 32. However, we do not provide an algorithm to decide the subset property in general. Again, any regular sort  $S^M$  can be expressed as  $[x := S^M] x$ .
3.  $\sigma_1 u_1 \cup \dots \cup \sigma_n u_n$  with  $\sigma_i$  arbitrary,  $u_i \in T$  for some set  $T$  such that for any  $u, u' \in T$ ,  $\beta = mgu(u, u')$  is always pseudolinear if it exists, and again  $\beta u \in T$ .  
 The intersection can be computed using Thms. 31, and 51. Inhabitation can be decided as in class 2., but again we do not provide an algorithm to decide subsort in general. If we

take  $T$  to be the set of all constructor terms in which variables occur only at a fixed unique depth  $n$ , the requirements to  $T$  are fulfilled, and each regular sort can be expressed.

As shown in section 4, dependent regular t-sets can express certain relations between distinct variables, e.g. the conditional equation  $x^{:\text{Nat}} < y^{:\text{Nat}} \rightarrow f(x, y) = g(x, y)$  can be expressed unconditionally by  $f(x, y) = g(x, y)$ , where the value combinations of  $x$  and  $y$  are restricted by  $\text{Nat}_{x < y}$  from Fig. 7. Since we have the problem that Thms. 31 and 34 use factorization  $\sigma/\beta$  which is not always a regular t-set, cf. Ex. 53, we have to restrict  $T$  as above. It is an unsolved problem whether a superclass of 2. exists that allows dependent t-sets but is still closed wrt. the required operations, especially intersection.

All classes follow the philosophy of allowing arbitrary nonlinearities up to a finite depth and forbidding any below. Since class 1. is sufficient to represent the set of all possible values  $U$  of an arbitrary constructor term  $u$ , we will use this class in the rest of this paper. In classical order-sorted approaches, each variable in a term is assigned a sort, e.g.  $x^{:\text{Nat}} + x^{:\text{Nat}}$ . We will, instead, use a semi-independent t-set to specify the set of possible ground instances, written e.g.  $^{(\text{Nat}_x)}(x + x)$ , with the informal meaning that each (“admissible”) substitution instantiating  $x + x$  must be extendable to a ground substitution contained in  $\text{Nat}_x^M$ . This approach still allows variable bindings in a term to be reflected by its sort, as sketched in Sect. 1; what we lose is the possibility of expressing nontrivial relations between variables.

**Definition 56.** We define an annotated term as a pair of a semi-independent regular t-set  $\sigma$  and an (unsorted) term  $v$ ; it is written as  $^\sigma v$ . The t-set  $\sigma$  denotes the admitted instances of  $v$ , cf. the use of  $^\sigma v$  in Defs. 61 and 74 below.

**Definition 57.** We call an expression of the form  $\sigma u$  with semi-independent regular  $\sigma$  and  $u \in \mathcal{TCR}, \nu$  an extended sort. The set of all possible values  $U$  of an annotated term  $^\sigma u$  is always an extended sort, viz.  $U = \sigma u$ . Sets of the form  $\sigma v$ , where  $v$  contains non-constructor functions, will be approximated by extended sorts later, cf. Sect. 6.

**Corollary 58.** A sorted equation  $^\sigma u_1 = ^\sigma u_2$  between annotated constructor terms is solvable iff an (unsorted) mgu  $\beta$  of  $u_1$  and  $u_2$  exists and  $\sigma/\beta \neq \{\}$ . The latter set contains the admissible ground instances of variables in  $\text{ran}(\beta)$ . An equation system  $^{\sigma_1} u_1 = ^{\tau_1} u'_1 \wedge \dots \wedge ^{\sigma_n} u_n = ^{\tau_n} u'_n$  can be reduced to a single equation  $^\sigma \langle u_1, \dots, u_n \rangle = ^\sigma \langle u'_1, \dots, u'_n \rangle$  by defining  $\sigma := \diamond_{i=1}^n \sigma_i \diamond \tau_i$ .

**Example 59.** Using the definitions from 53,  $\sigma \langle x, y \rangle \cap \top_{\{x'', y''\}} \langle x'', \text{cr}(y'', x'') \rangle$  cannot be represented as an extended sort.

*Proof.* Observe that  $\beta' = \beta \circ [x'' := x', y'' := y'] = \text{mgu}(\langle x, y \rangle, \langle x'', \text{cr}(y'', x'') \rangle)$ , and  $(\sigma \diamond \top_{\{x'', y''\}})/\beta' = \sigma/\beta$ ; hence, by 31,  $\sigma \langle x, y \rangle \cap \top_{\{x'', y''\}} \langle x'', \text{cr}(y'', x'') \rangle = (\sigma \diamond \top_{\{x'', y''\}})/\beta' \langle x', \text{cr}(y', x') \rangle = \sigma/\beta \langle x', \text{cr}(y', x') \rangle$ , the latter cannot be written as  $\tau_1 u_1 \cup \dots \cup \tau_n u_n$  with  $\tau_i$  regular, by an argument similar to 53.5.

## 6 Equational Theories

In this section, we extend the previous formalism to allow equationally defined functions  $f$ . We allow defining equations of the form given in Def. 60, thus ensuring the “executability” of  $f$ . Signatures

of such a function are computed from its defining equations by the *rg* algorithm presented below in Alg. 73, which will play a central role in pruning the search space of narrowing. The algorithm takes a regular t-set and a term with non-constructor functions and computes an upper approximation by an extended sort, e.g.  $rg([x, y := Nat], x + y) = [z := Nat] z$ . In terms of Sect. 1, we have  $rg(\sigma, v) = \overline{V}$  where  $\sigma$  denotes the values over which the variables in  $v$  may range. The *rg* algorithm consists of local transformations like rewriting and some simplification rules (cf. Def. 64), global transformations looking at a sequence of local transformation steps and recognizing certain kinds of self-references (cf. Lemma 67), and an approximation rule. Only the main rules can be discussed here; the complete algorithm is given in [4].

In Theorem 75, a narrowing calculus from [9] is equipped with sorts. In [4], the calculus is shown to remain complete if the applicability of its main rule is restricted by the disjointness test from Sect. 1.

**Definition 60.** In the rest of this section, we assume that  $f$  has the following defining equations:

$$\begin{aligned} \mu^1 f(u_{11}, \dots, u_{1n}) &= \mu^1 v_1, \\ &\dots, \\ \mu^m f(u_{m1}, \dots, u_{mn}) &= \mu^m v_m, \end{aligned}$$

where  $vars(v_i) \subset vars(u_{i1}, \dots, u_{in})$ . We assume that the variables of different defining equations are disjoint. Define  $dom(f, I) := \bigcup_{i \in I} \mu_i \langle u_{i1}, \dots, u_{in} \rangle$ , and  $dom(f) := dom(f, \{1, \dots, m\})$ .

**Definition 61.** Define the rewrite relation induced by the defining equations by:  $\sigma v_1 \rightarrow \sigma v_2$  iff

1. a defining equation  $\mu f(u_1, \dots, u_n) = \mu v$ , a substitution  $\beta$ , and a term  $v'(x)$  linear in  $x$  exist such that  $v_1 = v'(\beta f(u_1, \dots, u_n))$ ,  $v_2 = v'(\beta v)$ ,
2. and for all  $\sigma' \in \sigma$  there exists  $\mu' \in \mu$  such that for all  $x \in vars(u_1, \dots, u_n)$   $\epsilon \sigma' \beta x \rightarrow^* \epsilon \mu' x$  if  $\epsilon \sigma' \beta x$  is well-defined.

While the former condition is merely rewriting by pattern matching, the latter is an analogue to the classical well-sortedness requirement for  $\beta$ , requiring any well-defined variable instance to be admitted by the defining equation's sort. A ground term is called well-defined if it is reducible to a ground constructor term.  $\rightarrow^*$  and  $\leftrightarrow^*$  are defined as usual; the definition of  $\rightarrow$  is recursive, but well-founded. We require confluence and termination of  $\rightarrow$ , ensuring  $\mathcal{T}_{\mathcal{CR}} \subset \mathcal{T}_{\mathcal{CR}, \mathcal{F}} / \leftrightarrow^*$ , where  $\mathcal{T}_{\mathcal{CR}, \mathcal{F}} / \leftrightarrow^*$  denotes the set of equivalence classes of terms in  $\mathcal{T}_{\mathcal{CR}, \mathcal{F}}$  modulo  $\leftrightarrow^*$ . In other words,  $\leftrightarrow^*$  does not identify terms in  $\mathcal{T}_{\mathcal{CR}}$ , but new irreducible terms like  $nil + nil$  may arise which we will regard as “junk terms” and exclude from equation solutions. For a well-defined ground term  $v$ , let  $nf(v) \in \mathcal{T}_{\mathcal{CR}}$  denote its unique normal form; for  $A \subset \mathcal{T}_{\mathcal{CR}, \mathcal{F}}$ , let  $nf[A] := \{nf(v) \mid v \in A, v \text{ well-defined}\}$ .

Figure 10 shows a comparison of classical order-sorted terms and annotated terms. The applicability of  $\rightarrow$  is not decidable in general owing to the well-sortedness condition 61.2. It is possible to compute sufficiently large t-sets  $\mu_i$  for the defining equations such that 61.2 becomes trivial, cf. Alg. 71; however, if the  $\mu_i$  are too large, well-sorted terms arise that are not well-defined. As in any order-sorted term rewriting approach, we cannot overcome both problems simultaneously.

Range sorts are computed using expressions of the form  $(w_1 : u_1) \dots (w_n : u_n)$ , which can intuitively be thought of as generalized equation systems; the semantic is the set of all t-substitutions making each  $u_i$  equal ( $\leftrightarrow^*$ ) to an element of  $w_i$ . For example,  $(Nat : x)$  denotes  $\{[x := s^i(0)] \mid i \in \mathbb{N}\}$ , and  $(x + x : z)$  can be evaluated to  $\{[x := s^i(0), z := s^{2*i}(0)] \mid i \in \mathbb{N}\}$ .

	Classical order-sorted terms	Annotated terms
term	$w$ where $x_1^{:s_1}, \dots, x_m^{:s_m}$	$\sigma w$ where $\text{dom}(\sigma) = \{x_1, \dots, x_m\}$
sort	$\text{sortof}(w)$	$\sigma w$
def. eq.	$f(l_1, \dots, l_n) = r$ where $y_1^{:t_1}, \dots, y_m^{:t_m}$	${}^\mu f(l_1, \dots, l_n) = {}^\mu r$ where $\text{dom}(\mu) = \{y_1, \dots, y_m\}$
rewriting	$v(\beta f(l_1, \dots, l_n)) \rightarrow v(\beta r)$ where $\forall y \in V \text{ sortof}(\beta y) \subset \text{sortof}(y)$	$\sigma v(\beta f(l_1, \dots, l_n)) \rightarrow \sigma v(\beta r)$ where $\forall \sigma' \in \sigma \exists \mu' \in \mu \forall y \in V \text{ } {}^\varepsilon \sigma' \beta y \rightarrow {}^\varepsilon \mu' y$
equation	$w_1 = w_2$ where $x_1^{:s_1}, \dots, x_m^{:s_m}$	$\sigma w_1 = \sigma w_2$
solution	$\gamma w_1 \leftrightarrow^* \gamma w_2$ where $\forall x \in W \text{ sortof}(\gamma x) \subset \text{sortof}(x)$	$\gamma w_1 \leftrightarrow^* \gamma w_2$ and $\tau$ where $\text{nf}[\tau \gamma \langle x_1, \dots, x_m \rangle] \subset \sigma \langle x_1, \dots, x_m \rangle$ and $\text{nf}[\tau \beta \langle u_1, \dots, u_n \rangle] \neq \{\}$

$w, w_1, w_2 \in \mathcal{TCR}, \mathcal{F}, \mathcal{V}$ ,  $l_1, \dots, l_n \in \mathcal{TCR}, \mathcal{V}$ ,  $\sigma, \tau, \mu \in \mathcal{T}_{(\mathcal{V} \mapsto \mathcal{CR})}^*$ ,

$W = \text{vars}(w) = \text{vars}(w_1, w_2) = \{x_1, \dots, x_m\} = \text{dom}(\sigma)$ ,

$V = \text{vars}(l_1, \dots, l_n) = \{y_1, \dots, y_m\}$

Examples:

	Classical order-sorted terms	Annotated terms
term	$x^{:\text{Nat}} + x^{:\text{Nat}}$	$[x = \text{Nat}] x + x$
sort	$\text{Nat} + \text{Nat} = \text{Nat}$	$[x := \text{Nat}] (x + x) = \text{Even}$
def. eq.	$a^{:\text{Nat}} + 0 = a^{:\text{Nat}}$ $a^{:\text{Nat}} + s(b^{:\text{Nat}}) = s(a^{:\text{Nat}} + b^{:\text{Nat}})$ $s(a^{:\text{Nat}}) + b^{:\text{Nat}} = s(a^{:\text{Nat}} + b^{:\text{Nat}})$	$[a = \text{Nat}] a + 0 = [a = \text{Nat}] a$ $[a, b = \text{Nat}] a + s(b) = [a, b = \text{Nat}] s(a + b)$ $[a, b = \text{Nat}] s(a) + b = [a, b = \text{Nat}] s(a + b)$
rewriting	$s(x^{:\text{Nat}} + x^{:\text{Nat}} + s(y^{:\text{Nat}}))$ $\rightarrow s(s(x^{:\text{Nat}} + x^{:\text{Nat}} + y^{:\text{Nat}}))$ where $\beta = [a := x + x, b := y]$ $\text{sortof}(\beta a) = \text{Nat} + \text{Nat} = \text{sortof}(a)$ $\text{sortof}(\beta b) = \text{Nat} = \text{sortof}(b)$	$[x, y = \text{Nat}] s(x + x + s(y))$ $\rightarrow [x, y = \text{Nat}] s(s(x + x + y))$ where $\beta = [a := x + x, b := y]$ for $[x := s^i(0), y := s^j(0)]$ choose $[x := s^{2i}(0), y := s^j(0)]$
equation	$s(s(x^{:\text{Nat}})) = y^{:\text{Nat}} + y^{:\text{Nat}}$	$[x, y = \text{Nat}] s(s(x)) = y + y$
solution	$\gamma = [x := z^{:\text{Nat}} + z^{:\text{Nat}}, y := s(z^{:\text{Nat}})]$ $\text{sortof}(\gamma x) = \text{Nat} + \text{Nat} = \text{sortof}(x)$ $\text{sortof}(\gamma y) = s(\text{Nat}) \subset \text{sortof}(y)$	$\gamma = [x := z + z, y := s(z)]$ and $\tau = [z := \text{Nat}]$ , $\text{nf}[\tau \gamma \langle x, y \rangle]$ $= \{ \langle s^{2i}(0), s^{i+1}(0) \rangle \mid i \in \mathbb{N} \}$ $\subset \langle \text{Nat}, \text{Nat} \rangle$

**Fig. 10.** Comparison of classical order-sorted terms and annotated terms

**Definition 62.** Define  $w^M \subset \mathcal{T}_{\mathcal{CR}, \mathcal{F}, \mathcal{V}}$  by:

$$\begin{aligned} S^M &:= S^M \text{ as in Def. 4} && \text{for } S \in \mathcal{S} \\ g(v_1, \dots, v_n)^M &:= \{g(v'_1, \dots, v'_n) \mid v'_1 \in v_1^M, \dots, v'_n \in v_n^M\} && \text{for } g \in \mathcal{CR} \cup \mathcal{F} \\ x^M &:= \{x\} && \text{for } x \in \mathcal{V} \end{aligned}$$

For  $w \in \mathcal{T}_{\mathcal{CR}, \mathcal{S}}$ ,  $w^M$  agrees with Def. 4. For  $w \in \mathcal{T}_{\mathcal{CR}, \mathcal{F}, \mathcal{V}}$  we always have  $w^M = \{w\}$ . We tacitly extend the operations of Sect. 3 to  $\mathcal{T}_{\mathcal{CR}, \mathcal{F}, \mathcal{V}}$  by treating function symbols from  $\mathcal{F}$  like constructors from  $\mathcal{CR}$ , e.g.  $(0_x) (x + x) = \{0 + 0\}$ .

Let  $(w : u)^M := \{\sigma' \mid \text{dom}(\sigma') = \text{vars}(w, u), \exists w' \in w^M, u' \in u^M \ \sigma' w' \leftrightarrow^* \sigma' u'\}$  and  $((w_1 : u_1) (w_2 : u_2))^M := (w_1 : u_1)^M \diamond (w_2 : u_2)^M$ . We write  $(\sigma)$  to denote an expression  $(w_1 : u_1) \dots (w_n : u_n)$  such that  $((w_1 : u_1) \dots (w_n : u_n))^M = \sigma$ , e.g.  $(\text{Nat}_{x,y})$  denotes  $(\text{Nat} : x)(\text{Nat} : y)$ , but note that  $\sigma$  need neither be independent nor even regular. The terms are unsorted in order to deal with t-sets explicitly;  $(\sigma w : \tau u)$  can be written as  $(\sigma) (\tau) (w : u)$ . Note that the t-substitutions in  $(w : u)^M$  always yield ground constructor terms.

**Lemma 63.** Let  $\varepsilon f(w_1, \dots, w_n) \rightarrow^* \varepsilon u$ , for  $w_1, \dots, w_n \in \mathcal{T}_{\mathcal{CR}, \mathcal{F}}$  and  $u \in \mathcal{T}_{\mathcal{CR}}$ ; let  $I \subset \{1, \dots, m\}$  such that  $nf[\top \langle w_1, \dots, w_n \rangle] \cap \text{dom}(f, I) = nf[\top \langle w_1, \dots, w_n \rangle] \cap \text{dom}(f)$ ; then,  $i \in I$  and  $\mu'_i \in \mu_i$  exists such that  $w_j \rightarrow^* \mu'_i u_{ij}$  for  $j = 1, \dots, n$  and  $\mu'_i v_i \rightarrow^* u$ .

*Proof.* Consider the first reduction at root position within the chain  $\varepsilon f(w_1, \dots, w_n) \leftrightarrow^* \varepsilon u$ .

**Definition 64.** The following local transformation rules for  $rg$  are defined (excerpt):

1.  $(f(v'_1, \dots, v'_n) : u) = \bigsqcup_{i \in I} (v_i : u) (v'_1 : u_{i1}) \dots (v'_n : u_{in}) (\mu_i)$   
if  $I \subset \{1, \dots, m\}$  arbitrary such that  $nf[\top \langle w_1, \dots, w_n \rangle] \cap \text{dom}(f, I) = nf[\top \langle w_1, \dots, w_n \rangle] \cap \text{dom}(f)$ , cf. the remarks on page 36.
2.  $(u' : x) (v : u) = (u' : x) ([x := u'] v : [x := u'] u)$  if  $x \notin \text{vars}(u')$ ,
3.  $(S : x) (S' : x) = (S \cap S' : x)$ ,

Rules 2. and 3. also show “ $(\cdot : \cdot)$ ” as a generalization of term equality and sort membership, respectively. All local rules satisfy the correctness criterion  $lhs^M|_{\text{vars}(lhs)} = rhs^M|_{\text{vars}(rhs)}$ .

*Proof.* Use 63 for correctness of rule 1.; correctness of 2. and 3. follows by simple computations.

Only one proper approximation rule is needed, viz.  $(w_1 : u_1) \dots (w_n : u_n)^M \subset (w_2 : u_2) \dots (w_n : u_n)^M$ ; all other rules can be made exact by including the left-hand side in the right-hand side.

Applying local transformations creates a computation tree with alternatives (separated by “|”) as nodes, each alternative having a unique computation path from the root, cf. Fig. 13. Global transformations operate on such computation trees. A proof methodology (“rank induction”) is provided in Def. 65 and Lemma 66 for their verification that also allows the introduction of new global rules, if necessary, for some class of applications.

**Definition 65.** Let  $\sigma' \in ((w_1 : u_1) \dots (w_n : u_n))^M$ , then e.g.  $\sigma' w_1 \leftrightarrow^* \sigma' u_1$ , where  $\sigma' u_1 \in \mathcal{T}_{\mathcal{CR}}$  is in normal form. Owing to confluence and termination, each “ $\rightarrow$ ” chain starting from  $\sigma' w_1$  ends after finitely many steps at  $\sigma' u_1$ .

Define  $\text{rank}(\sigma', (w_1 : u_1))$  as the length of the longest such chain, which always exists. Define  $\text{rank}(\sigma', (w_1 : u_1) \dots (w_n : u_n)) := \sum_{i=1}^n \text{rank}(\sigma', (w_i : u_i))$ . We always have  $\text{rank}(\sigma', (\sigma)) \in \mathbb{N}$  and  $\text{rank}(\sigma', (\tau)) = 0$  for  $\tau$  regular t-set.

**Lemma 66.** Let  $(\sigma) = (\sigma_1) \mid \dots \mid (\sigma_m)$  be the result of repeated application of the rules from Def. 64, let  $z \in \text{dom}(\sigma) \cap \text{dom}(\sigma_1) \cap \dots \cap \text{dom}(\sigma_m)$ . Then for each  $\sigma' \in (\sigma)^M$ , an  $i \in \{1, \dots, m\}$  and a  $\sigma'_i \in (\sigma_i)^M$  exists such that  $\sigma'z = \sigma'_iz$  and  $\text{rank}(\sigma', (\sigma)) \geq \text{rank}(\sigma'_i, (\sigma_i)) + n_i$ , where  $n_i$  denotes the number of applications of Def. 64.1 in the path from  $(\sigma)$  to  $(\sigma_i)$ .

*Proof.* Induction on the number of applications of rules from Def. 64.

**Lemma 67.** (Global Transformation: Loop-Checking Rule)

Assume  $z \notin \text{dom}(\sigma) \supset \text{vars}(v) \not\equiv x$  and a computation tree of the form

$$\begin{aligned} (\sigma) (v : z) &= \dots \\ &= (\sigma) (v : x) (u_1(x) : z) \mid \dots \mid (\sigma) (v : x) (u_n(x) : z) \mid (u_{n+1} : z) \mid \dots \mid (u_m : z) \end{aligned}$$

where in each alternative's path at least one application of rule 64.1 occurred. Then,  $((\sigma) (v : z))^M \subset (S : z)^M$ , where  $S$  is a new sort name defined by  $S \doteq u_1(S) \mid \dots \mid u_n(S) \mid u_{n+1} \mid \dots \mid u_m$ . If all  $u_i(x)$  are linear in  $x$ , we have equality.

*Proof.* Show  $\sigma' \in ((\sigma) (v : z))^M \Rightarrow \sigma'z \in s^M$  by induction on  $\text{rank}(\sigma', (\sigma) (v : z))$ , using Lemma 66.

**Lemma 68.** Let  $1 \leq k \leq n \leq m$ ,

$$\begin{aligned} \text{assume } & (\sigma) (v : z) \\ &= (\sigma) (v : u_1) (u'_1 : z) \mid \dots \mid (\sigma) (v : u_n) (u'_n : z) \\ & \mid (\sigma_{n+1}) (v_{n+1} : u_{n+1}) (u'_{n+1} : z) \mid \dots \mid (\sigma_m) (v_m : u_m) (u'_m : z) \end{aligned}$$

where in each alternative's path at least one application of Def. 64.1 occurred.

$$\begin{aligned} \text{Then, } & (\sigma) (v : z) \\ &= (\sigma) (v : u_{k+1}) (u'_{k+1} : z) \mid \dots \mid (\sigma) (v : u_n) (u'_n : z) \\ & \mid (\sigma_{n+1}) (v_{n+1} : u_{n+1}) (u'_{n+1} : z) \mid \dots \mid (\sigma_m) (v_m : u_m) (u'_m : z) \end{aligned}$$

provided  $(u'_i : u_j)^M = \{\}$  for all  $i \in \{k+1, \dots, m\}, j \in \{1, \dots, k\}$ .

Intuitively, constructor terms  $u_1, \dots, u_k$  may be produced as the value of  $z$  or  $v$  only in alternatives  $1, \dots, k$ , but this in turn requires a constructor term  $u_1, \dots, u_k$ . Hence, there is no recursion basis, i.e.  $v$  may not have a  $u_i$  as its value, i.e. the first  $k$  alternatives are superfluous.

*Proof.* Show  $\sigma' \in ((\sigma) (v : z))^M \Rightarrow \bigwedge_{j=1}^k \forall \tau' \in \top \sigma'z \neq \tau'u_j$  by induction on  $\text{rank}(\sigma', (\sigma) (v : z))$ , using 66.

**Lemma 69.** Let  $y, x_1, \dots, x_k \in \mathcal{V}$ ,  $w \in \mathcal{TCR}_{\mathcal{F}, \mathcal{V}}$ ,  $u(x_1, \dots, x_k) \in \mathcal{TCR}_{\{x_1, \dots, x_k\}}$ ,  $u_{ij}, v_{ij}, v_i \in \mathcal{TCR}_{\mathcal{V}}$ ,  $v_i(y) \in \mathcal{TCR}_{\{y\}}$  for  $1 \leq i \leq n_1$ ,  $v_i \in \mathcal{TCR}$  for  $n_1 + 1 \leq i \leq n_2$ ,

We abbreviate  $u(y, u_{12}, \dots, u_{1k})$  to  $u(y, \mathbf{u}_1)$ .

Assume



$$\begin{array}{l}
(\sigma) \quad (\tau) \quad (w : z) \\
= (\beta_1 \sigma) \quad (\tau_1) \quad (\beta_1 w : u(y, \mathbf{u}_1)) \quad (u(v_1(y), \mathbf{v}_1) : z) \\
| \dots \\
| (\beta_{n_1} \sigma) \quad (\tau_{n_1}) \quad (\beta_{n_1} w : u(y, \mathbf{u}_{n_1})) \quad (u(v_{n_1}(y), \mathbf{v}_{n_1}) : z) \\
| (\beta_{n_1+1} \sigma) \quad (\tau_{n_1+1}) \quad (\beta_{n_1+1} w : u_{n_1+1}) \quad (u(v_{n_1+1}, \mathbf{v}_{n_1+1}) : z) \\
| \dots \\
| (\beta_{n_2} \sigma) \quad (\tau_{n_2}) \quad (\beta_{n_2} w : u_{n_2}) \quad (u(v_{n_2}, \mathbf{v}_{n_2}) : z) \\
| \quad (\tau_{n_2+1}) \quad (\beta_{n_2+1} w : u_{n_2+1}) \quad (v_{n_2+1} : z) \\
| \dots \\
| \quad (\tau_{n_3}) \quad (\beta_{n_3} w : u_{n_3}) \quad (v_{n_3} : z)
\end{array}$$

and  $(u_i : u(x_1, \mathbf{x})) = \perp_s = (v_j : u(x_1, \mathbf{x}))$  for  $n_1 + 1 \leq i \leq n_2$  and  $n_2 + 1 \leq j \leq n_3$ .  
Define  $S \doteq v_1(S) \mid \dots \mid v_{n_1}(S) \mid v_{n_1+1} \mid \dots \mid v_{n_2}$ .

Then,

$$\begin{array}{l}
(\sigma) \quad (\tau) \quad (w : z) \\
= (\beta_1 \sigma) \quad (\tau_1) \quad (\beta_1 w : u(y, \mathbf{u}_1)) \quad (u(v_1(y), \mathbf{v}_1) : z) \quad (S : y) \\
| \dots \\
| (\beta_{n_1} \sigma) \quad (\tau_{n_1}) \quad (\beta_{n_1} w : u(y, \mathbf{u}_{n_1})) \quad (u(v_{n_1}(y), \mathbf{v}_{n_1}) : z) \quad (S : y) \\
| (\beta_{n_1+1} \sigma) \quad (\tau_{n_1+1}) \quad (\beta_{n_1+1} w : u_{n_1+1}) \quad (u(v_{n_1+1}, \mathbf{v}_{n_1+1}) : z) \\
| \dots \\
| (\beta_{n_2} \sigma) \quad (\tau_{n_2}) \quad (\beta_{n_2} w : u_{n_2}) \quad (u(v_{n_2}, \mathbf{v}_{n_2}) : z) \quad | \\
| \quad (\tau_{n_2+1}) \quad (\beta_{n_2+1} w : u_{n_2+1}) \quad (v_{n_2+1} : z) \\
| \dots \\
| \quad (\tau_{n_3}) \quad (\beta_{n_3} w : u_{n_3}) \quad (v_{n_3} : z)
\end{array}$$

Intuitively, a constructor term of the form  $u(v, \dots)$  can occur only in two places:

- in one of the alternatives  $1, \dots, n_1$ ,  $v$  having the form  $v_i(v')$  where  $u(v', \dots)$  occurred earlier; or
- in one of the alternatives  $n_1 + 1, \dots, n_2$ ,  $v$  having the form  $v_i$ .

Thus, it is always true that  $v \in S^M$ .

*Proof.* Show  $\exists \sigma' \in (\sigma)(w : z)^M \quad u(u'_1, \mathbf{u}') = \sigma' z \Rightarrow u'_1 \in S^M$  by induction along the order  $u(u'_1, \mathbf{u}') < u(u''_1, \mathbf{u}'') \Leftrightarrow u'_1 \triangleleft u''_1$ .

**Algorithm 70.** The following algorithm provides an initial, coarse approximation  $max_f$  of the range sorts for an equationally defined function  $f$ .

$$max_f \doteq max_{\mu_1, v_1} \mid \dots \mid max_{\mu_m, v_m}$$

where  $max_{\mu, w}$  for  $w \in \mathcal{TCR}, \mathcal{F}, \mathcal{V}$  is defined by:

$$\begin{aligned}
max_{\mu, g(w_1, \dots, w_n)} &:= max_g && \text{if } g \in \mathcal{F} \\
max_{\mu, cr(w_1, \dots, w_n)} &\doteq cr(max_{\mu, w_1}, \dots, max_{\mu, w_n}) && \text{if } cr \in \mathcal{CR} \\
max_{\mu, x} &\doteq apply(\mu, x) && \text{if } x \in \mathcal{V}
\end{aligned}$$

If  $f(w_1, \dots, w_n) \rightarrow^* u \in \mathcal{TCR}$ , then  $u \in max_f^M$ , as can be shown by induction on the length of the  $\rightarrow$  chain.

**Algorithm 71.** Assume  $f$  is defined by the (yet unsorted) equations

$$f(u_{11}, \dots, u_{1n_1}) = v_1$$

...

$$f(u_{m1}, \dots, u_{mn_m}) = v_m$$

Assume that for each  $f \in \mathcal{F}$  a set  $F_f \subset \mathcal{F}$  of admitted function symbols for arguments of  $f$  is given. The following algorithm finds minimal independent t-sets  $\mu_i$  such that the applicability of  $\rightarrow$  becomes trivial if only subterms starting with a  $g \in F_f$  appear at the argument positions of  $f$ .

$$\max'_f \doteq \max'_{f,v_1} \mid \dots \mid \max'_{f,v_m}$$

where  $\max'_{f,w}$  for  $w \in \mathcal{TCR}_{\mathcal{F},\mathcal{V}}$  is defined by:

$$\max'_{f,g(w_1,\dots,w_n)} := \max'_g \quad \text{if } g \in \mathcal{F}$$

$$\max'_{f,cr(w_1,\dots,w_n)} \doteq cr(\max'_{f,w_1}, \dots, \max'_{f,w_n}) \text{ if } cr \in \mathcal{CR}$$

$$\max'_{f,x} \doteq \bigvee_{g \in F_f} \max'_g \quad \text{if } x \in \mathcal{V}$$

Define  $\mu_i = \text{compose}(\{\text{abstract}(x, \max'_{f,x}) \mid x \in \text{vars}(u_{i1}, \dots, u_{in_i})\})$ .

Then,  $f(v_1, \dots, v_n) \rightarrow v$  iff 61.1 is satisfied and  $(v_i \in \mathcal{V} \text{ or } v_i = g(v_i) \text{ with } g \in \mathcal{CR} \cup \mathcal{F}_f)$ .

**Example 72.** For the functions defined by the unstarred equations of Fig. 15, allowing arbitrary argument terms for  $+$  and  $\text{dup}$ , but only constructor terms from  $\text{Bin}$  as arguments for  $\text{val}$ , one gets  $F_+ = F_{\text{dup}} = \{+, \text{dup}, \text{val}\}$ ,  $F_{\text{val}} = \{\}$ , and

$$\begin{array}{ll} \max'_+ \doteq \max'_{+,x} \mid s(\max'_+) & \max'_+ \doteq \text{Nat} \\ \max'_{+,x} \doteq \max'_+ \mid \max'_{\text{dup}} \mid \max'_{\text{val}} & \max'_{+,x} \doteq \text{Nat} \\ \max'_{\text{dup}} \doteq \max'_+ & \max'_{\text{dup}} \doteq \text{Nat} \\ \max'_{\text{val}} \doteq 0 \mid \max'_{\text{dup}} \mid s(\max'_{\text{dup}}) & \max'_{\text{val}} \doteq \text{Nat} \end{array}$$

or, simplified:

which corresponds to the implicit t-set shown in Fig. 15.

**Algorithm 73.** To compute  $rg(\sigma, v)$ , start with the expression  $(\sigma) (v : z)$ , where  $z$  is new, and repeatedly apply rules in the following order: global rules, approximation rule, simplifying local rules (like Defs. 64.2 and 64.3), and rewriting (Def. 64.1). Apply approximation only if certain conditions make it necessary; apply all other rules wherever possible. By setting certain parameters in the termination criterion, the trade-off between computation time and precision of the result can be controlled. On termination, an expression  $(\sigma_1) (u_1 : z) \mid \dots \mid (\sigma_n) (u_n : z)$  with regular t-sets  $\sigma_i$  is obtained. The final result is then  $rg(\sigma, v) := \sigma_1 u_1 \mid \dots \mid \sigma_n u_n$ , satisfying  $nf[\sigma^M v] \subset rg(\sigma, v)^M$ .

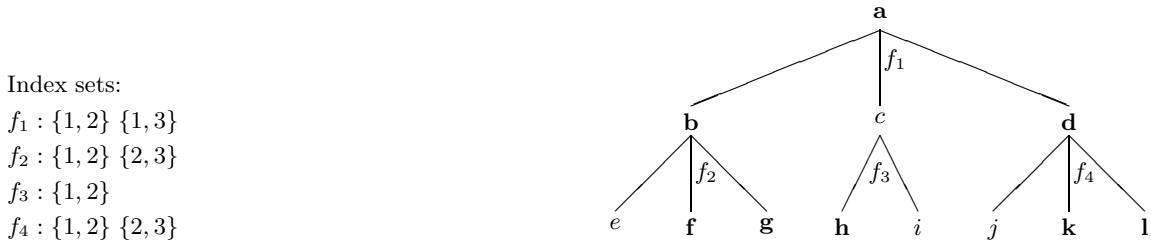
$$\begin{array}{ll} x + 0 & = x \\ x + s(y) & = s(x) + y \\ \\ (Nat : x) (Nat : y) (x + y : z) & \\ = (Nat : x) (Nat : y) (x : x_1) (y : 0) (x_1 : z) & 64.1 \\ \mid (Nat : x) (Nat : y) (x : x_1) (y : s(y_1)) (s(x_1) + y_1 : z) & \\ = (Nat : x) (x : z) & \text{simplification} \\ \mid (Nat : x_1) (Nat : y_1) (s(x_1) + y_1 : z) & \\ = (Nat : x) (x : z) & 64.1 \\ \mid (Nat : x_2) (x_2 : z) & + \text{ simplification} \\ \mid (Nat : x_1) (Nat : y_2) (s(s(x_1)) + y_2 : z) & \\ = \dots & \\ = (Nat : x) (x : z) & 64.1 \\ \mid (Nat : x) (Nat : y) (s^i(x) + y : z) & + \text{ simplification} \\ = \dots & \end{array}$$

**Fig. 11.** Nonterminating sort rewriting computation

The termination of Alg. 73 has to be artificially enforced. Certainly, the rewrite relation  $\rightarrow$  is Noetherian, i.e. each computation chain starting from a *term* will terminate. However, Alg. 73 computes with

*sorts* that represent infinitely many terms in general, and the length of their computation chains may increase unboundedly. Hence, *sort* rewriting need not terminate even though *term* rewriting terminates. As an example, consider the equational theory and the computation shown in Fig. 11.

In principle, Alg. 73 can be stopped after every step, using the approximation by  $max_f$ ; in other words, there is a trade-off between computation time and the precision of the result. We suggest the following termination criterion: applying Rule 64.1 to an expression  $(f(\dots) : \dots)$  is allowed only if less than  $\#use(dom(f))$  rewrite steps wrt.  $f$  have occurred in the current path<sup>7</sup>. The – heuristic – justification considers that  $f$  is defined recursively over the structure of  $dom(f)$  and that no more than  $\#use(dom(f))$  rewrite steps are necessary to “get back to the starting expression”, thus making e.g. Rule 67 applicable. Since all other local and global transformations except 64.1 can be applied only a finite number of times, this criterion ensures termination.



**Fig. 12.** Applying a global transformation in an example computation tree

The defining equations of a function  $f$  need not be independent in a logical / axiomatic sense; arbitrarily many “derived” equations may be added, cf. Fig. 13. Accordingly, it suffices to use only a subset of the equations for a rewrite step by 64.1, provided  $dom(f)$  is still completely covered (cf. the role of the index set  $I$  in 64.1). Since it is not possible to select a suitable  $I$  at the time the rewrite step is conducted we proceed the other way round: we use *all* equations for  $f$  in each rewrite step, making a global transformation applicable not only if all alternatives have the required form but even when a subset of alternatives has the required form and confinement to these alternatives still leads to index sets completely covering  $dom(f)$  in all relevant rewrite steps.

In this way, supplying additional derived function equations may result in making “better” global transformations applicable, and hence in enhancing the precision of the computed sort. Thus, we may get an effect similar to that obtained by term declarations in [12].

The test for applicability of a global transformation works as follows: for each alternative  $(\sigma)$  that does not meet the applicability criterion, delete all complete index sets in the last rewrite step leading to  $(\sigma)$ . If no index set remains, omit this rewrite step, and delete in turn all complete index sets in the previous rewrite step in the computation tree. If no previous rewrite step exists, the global transformation cannot be made applicable. If a complete index set still exists in the first, highest-level rewrite step after all deletions are done, the transformation has been made applicable.

As an example, consider the computation tree shown in Fig. 12. Rewrite steps have been conducted for functions  $f_1, f_2, f_3, f_4$ , transforming alternative  $a$  into  $b \mid c \mid d$ , and in turn to  $e \mid f \mid g \mid h \mid i \mid j \mid k \mid l$  (only applications of 64.1 are shown). The complete index sets for each function are listed in the table. Suppose a global transformation were applicable if we could restrict our attention to  $f \mid g \mid h \mid k \mid l$  (shown in bold face). The procedure described above results in selection of the

<sup>7</sup> For an extended sort  $S = \sigma u$ , we define  $use(S) = use(\sigma)$ .

index set  $\{2, 3\}$  for both,  $f_2$  and  $f_4$ , and  $\{1, 3\}$  for  $f_1$ , deleting alternative  $c$ , as well. Hence, the transformation has been made applicable.

Taking the definitions in Fig. 15, we can compute  $rg(Nat_x, x + x)$ :

$$\begin{aligned}
& (Nat : x) (x + x : z) \\
&= (Nat : x) (x_1 : z) (x : x_1) (x : 0) \mid (Nat : x) (x_1 : z) (x : 0) (x : x_1) \mid \\
&\quad (Nat : x) (s(x_1 + y_1) : z) (x : x_1) (x : sy_1) \mid (Nat : x) (s(x_1 + y_1) : z) (x : sx_1) (x : y_1) \\
&= (0 : z) \mid (Nat : y_1) (sy_1 + y_1 : z_1) (sz_1 : z) \mid (Nat : x_1) (x_1 + sx_1 : z_1) (sz_1 : z) \\
&= (0 : z) \mid (Nat : y_2) (ssy_2 + y_2 : z_2) (ssz_2 : z) \mid (Nat : y_2) (y_2 + y_2 : z_2) (ssz_2 : z) \\
&\quad \mid (Nat : y_2) (y_2 + y_2 : z_2) (ssz_2 : z) \mid (Nat : x_2) (x_2 + ssx_2 : z_2) (ssz_2 : z) \\
&= (0 : z) \mid (Nat : y_2) (y_2 + y_2 : z_2) (ssz_2 : z) \\
&= (Even : z)
\end{aligned}$$

where the new sort definition  $Even \doteq 0 \mid s(s(Even))$  is generated. The performed steps are: Rule 64.1 with equations a.-d.; simplification; Rule 64.1 with c.-d. twice in parallel, including simplification; deletion of the 2nd, 4th, and 5th alternative, since they are covered by the 3rd one; this makes Lemma 67 applicable as the final step.

**Fig. 13.** Range sort computation for  $x + x$

**Definition 74.** A substitution  $\beta$  is called a solution of an equation  $\sigma v_1 = \sigma v_2$  iff a t-set  $\tau$  exists that denotes the sorts of variables in the  $ran(\beta)$  such that

1.  $\tau\beta v_1 \leftrightarrow^* \tau\beta v_2$ ,
2.  $\forall \tau' \in \tau \ \exists \sigma' \in \sigma \ \forall x \in vars(v_1, v_2) \ \tau'\beta x \text{ well-defined} \Rightarrow \tau'\beta x \leftrightarrow^* \sigma' x$ ,  
or equivalently:  $nf[\tau\beta\langle x_1, \dots, x_n \rangle] \subset \sigma\langle x_1, \dots, x_n \rangle$ , where  $\{x_1, \dots, x_n\} = vars(v_1, v_2)$ ,  
similar to the classical well-sortedness requirement for  $\beta$ , and
3.  $nf[\tau\beta v_1] \neq \{\}$ , i.e. the solution has at least one well-defined ground instance.

**Theorem 75.** An arbitrary narrowing calculus preserving solution sets remains complete if restricted appropriately by sorts. For example, for lazy narrowing [9], abbreviating  $\tau := \sigma \diamond \top_{vars(u_1, \dots, u_n)}$ , we get for the main rules:

$$\begin{aligned}
\text{(ln)} \quad & \frac{\tau v_1 = \tau u_1 \wedge \dots \wedge \tau v_n = \tau u_n \wedge \tau v = \tau v'}{\sigma f(v_1, \dots, v_n) = \sigma v} & \begin{array}{l} \top f(u_1, \dots, u_n) = \top v' \text{ defining equation} \\ inh(inf(rg(\tau, v), rg(\tau, v'))), \\ inh(inf(rg(\tau, v_1), rg(\tau, u_1))), \dots \\ inh(inf(rg(\tau, v_n), rg(\tau, u_n))) \end{array} \\
\text{(d)} \quad & \frac{\tau u_1 = \tau v_1 \wedge \dots \wedge \tau u_n = \tau v_n}{\sigma f(u_1, \dots, u_n) = \sigma f(v_1, \dots, v_n)} & \begin{array}{l} inh(inf(rg(\tau, u_1), rg(\tau, v_1))), \dots, \\ inh(inf(rg(\tau, u_n), rg(\tau, v_n))) \end{array}
\end{aligned}$$

In rule (ln), the remaining equations  $\tau v_1 = \tau u_1, \dots, \tau v_n = \tau u_n$  can often be solved by purely syntactic unification. In this case, the non-disjointness criteria  $inh(inf(rg(\tau, v_1), rg(\tau, u_1))), \dots, inh(inf(rg(\tau, v_n), rg(\tau, u_n)))$  are trivially satisfied and may be omitted in practical implementations. Note that the variables in defining equations have to be assigned the sort  $\top$ . Starting from a conditional narrowing calculus, nontrivially sorted defining equations become possible.

$$\begin{aligned}
& (Bin : x) (val \cdot x : z) \\
= & (Bin : x) (x : nil) (0 : z) && \text{Def. } val \\
& | (Bin : x) (x : x_1 :: o) (dup \cdot val \cdot x_1 : z) \\
& | (Bin : x) (x : x_1 :: i) (s \cdot dup \cdot val \cdot x_1 : z) \\
= & (0 : z) \\
& | (Bin : x_1) (dup \cdot val \cdot x_1 : z) \\
& | (Bin : x_1) (dup \cdot val \cdot x_1 : z_1) (s \cdot z_1 : z) \\
= & (0 : z) && \text{Def. } dup \\
& | (Bin : x_1) (val \cdot x_1 : 0) (0 : z) \\
& | (Bin : x_1) (val \cdot x_1 : s \cdot x_2) (s \cdot s \cdot dup \cdot x_2 : z) \\
& | (Bin : x_1) (val \cdot x_1 : 0) (0 : z_1) (s \cdot z_1 : z) \\
& | (Bin : x_1) (val \cdot x_1 : s \cdot x_2) (s \cdot s \cdot dup \cdot x_2 : z_1) (s \cdot z_1 : z) \\
= & (0 : z) \\
& | (Bin : x_1) (val \cdot x_1 : s \cdot x_2) (dup \cdot x_2 : z_2) (s \cdot s \cdot z_2 : z) \\
& | (Bin : x_1) (val \cdot x_1 : 0) (s \cdot 0 : z) \\
& | (Bin : x_1) (val \cdot x_1 : s \cdot x_2) (dup \cdot x_2 : z_2) (s \cdot s \cdot s \cdot z_2 : z) \\
\stackrel{(\subseteq)}{=} & (0 : z) && (*) \\
& | (Bin : x_1) (max_{val} : s \cdot x_2) (dup \cdot x_2 : z_2) (s \cdot s \cdot z_2 : z) \\
& | (Bin : x_1) (max_{val} : 0) (s \cdot 0 : z) \\
& | (Bin : x_1) (max_{val} : s \cdot x_2) (dup \cdot x_2 : z_2) (s \cdot s \cdot s \cdot z_2 : z) \\
= & (0 : z) && (**) \\
& | (Nat : s \cdot x_2) (dup \cdot x_2 : z_2) (s \cdot s \cdot z_2 : z) \\
& | (Nat : 0) (s \cdot 0 : z) \\
& | (Nat : s \cdot x_2) (dup \cdot x_2 : z_2) (s \cdot s \cdot s \cdot z_2 : z) \\
= & (0 : z) && dup, \text{ see above} \\
& | (s \cdot s \cdot Even : z) | (s \cdot 0 : z) | (s \cdot s \cdot s \cdot Even : z) \\
= & (Nat : z)
\end{aligned}$$

(\*): Here,  $val \cdot x_1$  is estimated upwards, since the original expression  $(val \cdot x_1 : \dots)$  occurs as part of the actual expression, but the introduction of a new recursive sort definition is prohibited by the presence of the non-constructor function  $dup$ .

We write “ $\stackrel{(\subseteq)}{=}$ ” to indicate that  $rg$  here differs from the real, semantic range sort.

(\*\*): One can trivially transform the defining equations of  $val$  and  $dup$  into sort definitions by replacing function applications with corresponding sort names. This yields an upper bound for the range sorts:

$$\begin{aligned}
max_{val} & \dot{=} 0 \mid s \cdot max_{dup} \mid max_{dup} && \text{and} \\
max_{dup} & \dot{=} 0 \mid s \cdot s \cdot max_{dup} \\
\text{i.e. } max_{dup}^M & = Even \text{ and } max_{val}^M = Nat^M
\end{aligned}$$

**Fig. 14.** Range sort computation for  $val$

*Proof.* Rules may be restricted using the fact that the solution set of  $\sigma_1 v_1 = \sigma_2 v_2$  is inhabited only if  $rg(\sigma_1, v_1)^M \cap rg(\sigma_2, v_2)^M \supset nf[\sigma_1^M v_1] \cap nf[\sigma_2^M v_2] \neq \{\}$ .

To prove the completeness of assigning sorts to variables in goal equations, observe that each definition of a regular t-set  $\sigma$  can be transformed into a definition of a function  $f_\sigma$  admitted by Def. 61 such that  $\sigma' \in \sigma^M$  iff  $f_\sigma(\sigma' x_1, \dots, \sigma' x_n) = true$ , where  $dom(\sigma) = \{x_1, \dots, x_n\}$  and  $true \in \mathcal{CR}$ . Hence, a sorted equation  $\sigma v_1 = \sigma v_2$  can be simulated without sorts by  $v_1 = v_2 \wedge f_\sigma(x_1, \dots, x_n) = true$ . Assigning non-trivial sorts to variables in defining equations is possible for conditional calculi in a similar way.

**Lemma 76.** Let  $\sigma$  be independent, and let  $x$  be new; then, the equation  $\sigma v_1 = \sigma v_2$  has a solution iff  $rg(\sigma, \langle v_1, v_2 \rangle)^M \cap \top \langle x, x \rangle \neq \{\}$ , provided the approximation rule was not used in  $rg$  computation.

Lemma 76 shows that the amount of search space reduction by the sorts depends only on the quality of approximations by  $rg$  and the expressiveness of our sort language. Without the reflection of variable bindings in sorts, such a result is impossible, even if no “occur check” and no non-constructor functions are involved, e.g.  $\langle x, y \rangle = \langle 0, s(0) \rangle$  is solvable, but  $\langle x, x \rangle = \langle 0, s(0) \rangle$  is not.

It is possible to extend the presented framework to cope with unfree constructors, too. This allows us, for example, to define a sort *Set* of sets of natural numbers, cf. App. B. As we show below, it is sufficient to be able to compute the closure of a sort wrt. the congruence relation induced by the equations between constructors.

**Definition 77.** Assume we are given certain equations between constructors in addition to the equations for defined functions. As in Def. 61, we define the rewrite relation  $\rightarrow$  to be induced by the equations between constructors. We do not require  $\rightarrow$  to be confluent, nor to be Noetherian. For the union of both equation sets, we similarly define  $\rightarrow$ . We require the defining equations to be compatible with the constructor equations, i.e.

$$\bigwedge_{i=1}^n v_i \leftrightarrow^* v'_i \implies nf(f(v_1, \dots, v_n)) \leftrightarrow^* nf(f(v'_1, \dots, v'_n))$$

whenever at least one of the two normal forms exists. Note that the sort algorithms work only on free sorts and hence ignore the relation  $\leftrightarrow^*$ .

**Lemma 78.** If  $v, w \in \mathcal{T}_{\mathcal{CR}, \mathcal{F}}$  are well-defined, we have  $v \leftrightarrow^* w \iff nf(v) \leftrightarrow^* nf(w)$ .

*Proof.* “ $\Leftarrow$ ” trivial; “ $\Rightarrow$ ” by induction on the length of the  $\leftrightarrow^*$  chain.

In each equivalence class wrt.  $\leftrightarrow^*$ , we may select an arbitrary element and declare it to be the normal form, thus defining  $nf_c$ . We adapt the notion of solution of an equation system from Def. 74 by replacing  $\leftrightarrow^*$  with  $\leftrightarrow^*$ , leaving condition 74.3 unchanged. Then, using Lemma 78, we can show that an equation  $\sigma v = \sigma v'$  has a solution only if  $nf_c[rg(\sigma, v)^M] \cap nf_c[rg(\sigma, v')^M] \neq \{\}$ . If we have an algorithm  $rg_c$  to compute upper approximations for  $nf_c[\cdot]$ , similar to  $rg$  for  $nf[\cdot]$ , we can extend the sorted narrowing rules from Def. 75 to cope with unfree constructors by replacing  $rg(\tau, v)$  with  $rg_c(rg(\tau, v))$ , etc. However, such an algorithm is not provided here.

## 7 Application in Formal Program Development

To support formal program development, we employ the paradigm of implementation proof, starting from an “abstract” operation  $ao$  on abstract data of sort  $as_1$  or  $as_2$  which are to be implemented by a corresponding “concrete” operation  $co$  on concrete data of sorts  $cs_1$  or  $cs_2$ , respectively. The connection between abstract and concrete data is established by representation func-

$$\begin{array}{ccc} as_1 & \xrightarrow{ao} & as_2 \\ r_1 \uparrow & & \uparrow r_2 \\ cs_1 & \xrightarrow{co} & cs_2 \end{array}$$

tions  $r_1 : cs_1 \rightarrow as_1$  and  $r_2 : cs_2 \rightarrow as_2$ , representing each concrete data term as an abstract one. Different concrete terms may represent the same abstract term. Thus, it is possible to perform the computation on the concrete level, and interpret the result on the abstract level. The correspondence between the concrete and abstract operation imposes correctness requirements on the concrete operation.

We wish to synthesize the concrete operation  $co$  as the Skolem function for  $y$  in the formula  $\forall \mathbf{x} \exists y \quad ao(r_1(\mathbf{x})) = r_2(y)$ . A suitable method for the constructive correctness proof is induction on the form of a data term  $\mathbf{x} \in cs_1$ , leading to a case distinction according to (one of) the head constructor(s) of  $\mathbf{x}$ . In each case, we have to solve an equation  $ao(r_1(\mathbf{x}_i)) = r_2(y)$  wrt.  $y$ . The synthesized function  $co$  is then given by equations  $co(\mathbf{x}_i) = \beta_i y$ , where  $\mathbf{x}_i$  is a data term starting with the  $i^{th}$  constructor, and  $\beta_i$  is the solving substitution for this case. After having solved an equation, one still has to check whether the solution  $\beta_i y$  is of the required sort  $cs_2$ , if not, a different solution must be found.

The sort discipline presented here supports specifically this method. Besides allowing recursive sort definitions of  $cs_1$ ,  $cs_2$ ,  $as_1$ , and  $as_2$  as well as recursive function definitions of  $ao$ ,  $r_1$ , and  $r_2$ , the induction principle from Thm. 9 provides the case distinction and proof goals for an induction on  $x \in cs_1^M$ . The sort discipline is able to cope with the additional problems of synthesis as compared with verification, i.e., to direct the construction of the solution term, to the extent that disjoint subsorts of a concrete sort are assigned with disjoint subsorts of the corresponding abstract sort. In this manner, the sort of an “abstract” term indicates which “concrete” terms are representing it.

As an example, consider the formal development of algorithms for binary numbers. Consider the sort and function definitions in Fig. 15. All terms are sorted by the t-set  $[x := Nat] \diamond [y := Nat] \diamond [z := Bin]$ , which is omitted in the equations for the sake of brevity. Equations marked by “\*” are redundant and can be proven by structural induction. The rightmost column contains the sort computed by  $rg$  for each equation; all sorts happen to be regular. We have axioms defining the “representation function”  $val : Bin \rightarrow Nat$ , and an auxiliary function  $dup$  to duplicate natural numbers which uses the addition  $+$  on natural numbers.

$Nat \doteq 0 \mid s(Nat)$		$a.$	$x + 0 = x$	$Nat$
$Bin \doteq nil \mid Bin::o \mid Bin::i$		$b.*$	$0 + x = x$	$Nat$
E.g.		$c.$	$x + s(y) = s(x + y)$	$s(Nat)$
$val(nil::i::o::i) = s^5(0)$		$d.*$	$s(x) + y = s(x + y)$	$s(Nat)$
Prove		$e.$	$dup(x) = x + x$	$Even$
$\forall c \exists z \quad s(val(c)) = val(z)$		$f.*$	$dup(x + y) = dup(x) + dup(y)$	$Even$
		$g.$	$val(nil) = 0$	$0$
		$h.$	$val(z::o) = dup(val(z))$	$Even$
		$i.$	$val(z::i) = s(dup(val(z)))$	$s(Even)$

**Fig. 15.** Sort and function definitions for synthesis of binary arithmetic algorithms

The main contribution of the sorts is the computation of  $rg(Nat_x, dup(x)) = [z := Even] \ z = Even$ , where the sort definition  $Even \doteq 0 \mid s(s(Even))$  is automatically introduced. Although only independent t-sets are involved in the example, the variable bindings in  $x + x$  are reflected by its sort, viz. *Even*. For this range-sort computation, the redundant equation *d*. is necessary, cf. Fig. 13.

Taking the easiest example, let us synthesize an algorithm *incr* for incrementing a binary number; the synthesis of algorithms for addition and multiplication is shown in App. A. The goal  $\forall c \exists z \ s(val(c)) = val(z)$  is proved by structural induction on *c*, the appropriate induction scheme being provided by Thm. 9, cf. Fig. 2. For example, in case  $c = c' :: o$ , we have to solve the equation  $s(val(c' :: o)) = val(z)$  wrt. *z*, and the sorted narrowing rule from Thm. 75 is only applicable to equation *i*. since the left-hand side's sort is computed as  $s(Even)$ . Note that, in order to get the full benefit of the sort calculus, narrowing should be applied only at the root of a term, since then additional sort information is supplied from the other side of the equation; this is the reason for using lazy narrowing. The employed calculus' drawback of admitting only trivially sorted defining equations is overcome by subsequently checking the solutions obtained for well-sortedness.

Narrowing with equation *h*. instead of *i*. would lead into an infinite branch<sup>8</sup>, trying to solve an equation  $s(dup(\dots)) = dup(\dots)$ . Such infinite branches are cut off by the sorts, especially by the global transformation rules which detect certain kinds of recursion loops. This seems to justify the computational overhead of sort computation. Thanks to the provided proof methodology based on regular t-sets, new global rules for detecting new recursion patterns can easily be added if required.

The control information provided by the sort calculus acquires particular importance in “proper” narrowing steps, i.e., the ones actually contributing to the solution term. While conventional narrowing procedures essentially enumerate each element of the constructor term algebra and test whether it is a solution, the presented sort calculus approaches the solutions directly, depending on the precision of computed range sorts.

The sort algorithms, especially *rg*, perform, in fact, simple induction proofs. For example, it is easy to prove by induction that  $x + x$  always has sort *Even*, and that sorts *Even* and  $s(Even)$  are disjoint, once these claims have been guessed or intuitively recognized. However, while a conventional induction prover would not propose these claims as auxiliary lemmas during the proof of  $\forall c \exists z \ s(val(c)) = val(z)$ , they are implicitly generated by the sort algorithms. The sort calculus allows the “recognition of new concepts”, so to speak, although only within the rather limited framework given by the sort language. In [8], an approach to the automatic generation of more complex auxiliary lemmas is presented based on E-generalization using regular sorts, too.

A prototype support system written in Quintus-Prolog takes a total of 41 seconds user time on a Sparc 1 to automatically conduct the 9 induction proofs, with 135 narrowing subgoals necessary for the development of incrementation, addition, and multiplication algorithms on binary numbers, cf. App. A. In the form of a paper case study from the area of compiler construction, an implementation of sets of lists of natural numbers by ordered son-brother trees has been proved, cf. App. B. The algorithm for inserting a new list into a tree is used to construct comb vectors for parse table compression; it is specified as an implementation of  $(\{\cdot\} \cup \cdot)$ . The use of sorts reduces the search space of the synthesis proof to that of a verification proof, i.e. it uniquely determines all proper narrowing steps or solution constructors. The computed signatures are too complex for there to be much likelihood of their being declared by a user who does not know the proof in advance.

---

<sup>8</sup> Cf. Figs. 19 and 21 in App. A, where the search space for this example is shown for both unsorted and sorted narrowing.



## 8 References

### References

1. V. Antimirov. Personal communication, Apr 1995.
2. Leo Bachmair and Harald Ganzinger. On restrictions of ordered paramodulation with simplification. In *Proc. 10th CADE*, volume 449 of *LNAI*, pages 427–441, Jul 1990.
3. Alexander Bockmayr. *Beiträge zur Theorie des logisch-funktionalen Programmierens*. PhD thesis, University Karlsruhe, 1991.
4. Jochen Burghardt. *Eine feinkörnige Sortendisziplin und ihre Anwendung in der Programmkonstruktion*. PhD thesis, Univ. Karlsruhe, 1993.
5. Hubert Comon. Equational formulas in order-sorted algebras. In *Proc. ICALP*, 1990.
6. R. Echahed. *On Completeness of Narrowing Strategies*, volume 298 of *LNCS*. Springer, 1988.
7. L. Fribourg. A narrowing procedure for theories with constructors. In *Proc. 7. CADE*, volume 170 of *LNCS*, pages 259–279, 1984.
8. Birgit Heinz. Lemma discovery by anti-unification of regular sorts. Technical Report 94–21, TU Berlin, 1994.
9. S. Hildobler. *Foundations of Equational Programming*, volume 353 of *LNAI*. Springer, 1989.
10. Eduard Klein and M. Martin. The parser generating system PGS. *Software Practice and Experience*, 19(11):1015–1028, 1989.
11. P. Mishra. Towards a theory of types in Prolog. In *Proc. 1984 International Symposium on Logic Programming*, pages 289–298. IEEE, 1984.
12. Manfred Schmidt-Schau. *Computational Aspects of an Order-Sorted Logic with Term Declarations*. PhD thesis, Univ. Kaiserslautern, Apr 1988.
13. J.W. Thatcher and J.B. Wright. Generalized finite automata theory with an application to a decision problem of second-order logic. *Mathematical Systems Theory*, 2(1), 1968.
14. M. Tommasi. Automates d’Arbres avec Tests d’egalites entre Cousins Germains. Technical report, LIFL-IT, 1991.
15. T.E. Uribe. Sorted unification using set constraints. In *Proc. CADE–11*, volume 607 of *LNCS*, pages 163–177, 1992.

# Appendix

## A Case Study “Binary Arithmetic”

In this appendix, the synthesis of algorithms for incrementation, addition, and multiplication of binary numbers is shown. Figure 16 gives an overview of the induction proofs conducted, together with the induction variable, the computation time (in seconds on a Sparc 1 under Quintus Prolog), and the number of subgoals.  $a$  and  $b$  denote (Skolem) constants, while  $x$  denotes a variable with respect to which the equation is to be solved. Note that in our paradigm, universally quantified variables are skolemized into constant symbols; induction is performed over just some of these constants. Figure 17 lists the synthesized algorithms.

On page 45, a protocol of the synthesis session is given. Pure induction proofs, i.e. ones that do not solve an equation wrt. some variable, are omitted. The notation in the Prolog implementation differs slightly from the one used in this paper. The predicate “*init\_sort\_system*” computes a range sort for every defining equation. The predicate “*solve*” tries to solve the given equation; every time the actual narrowing step is not uniquely determined by the sorts, the user is shown a menu and prompted to make a decision. During the session, the user always had to decide to start an induction, indicated by “**ind**”, optionally followed by a list of induction variables. Note that none of the proofs required any further user interaction. The execution trace shows the actual subgoal on entry into “*solve*”, and the solved subgoal together with the solving substitution on exit. At the end of the session, an example computation ( $5 * 6 = 30$ ) is performed (predicate “*eval\_term*”) using the newly synthesized algorithms, and the sort definitions (incomplete), proved laws, and function-defining equations are listed. Functions  $f_{24}$ ,  $f_{188}$ , and  $f_{429}$  compute the successor of a binary number, the sum of two binary numbers, and the product of two binary numbers, respectively.

Starting on page 50, the search space is shown in particular for the synthesis of the *incr* algorithm. Figures 18 to 20 show the search space in cases where no sorts are used to control narrowing; Fig. 21 shows the search space where sorts are used.

formula	ind. var.	time (sec.)	sub goals
initialization		3	
$s(val(a)) = val(x)$	$a$	5	10
$0 + a = a$	$a$	1	5
$s(a) + b = s(a + b)$	$b$	1	7
$dup(a) + dup(b) = dup(a + b)$	$b$	2	11
$val(a) + val(b) = val(x)$	$a, b$	19	47
$a + b + b = a + dup(b)$	$b$	2	11
$a * dup(b) = dup(a * b)$	$b$	2	13
$dup(val(a)) + val(b) = val(add(a :: o, b))$	$b$	3	18
$val(a) * val(b) = val(x)$	$b$	6	13
total		41	135

**Fig. 16.** Implementation proofs for binary arithmetic

$incr(nil)$	$= nil :: i$
$incr(x :: o)$	$= x :: i$
$incr(x :: i)$	$= incr(x) :: o$
$add(nil, nil)$	$= nil$
$add(nil, y :: o)$	$= y :: o$
$add(nil, y :: i)$	$= y :: i$
$add(x :: o, nil)$	$= x :: o$
$add(x :: o, y :: o)$	$= add(x, y) :: o$
$add(x :: o, y :: i)$	$= add(x, y) :: i$
$add(x :: i, nil)$	$= x :: i$
$add(x :: i, y :: o)$	$= add(x, y) :: i$
$add(x :: i, y :: i)$	$= incr(add(x, y)) :: o$
$mult(x, nil)$	$= nil$
$mult(x, y :: o)$	$= mult(x, y) :: o$
$mult(x, y :: i)$	$= add(mult(x, y) :: o, x)$

**Fig. 17.** Synthesized algorithms for binary arithmetic

## Synthesis session protocol

?- *init\_sort\_system.*

*val nil = 0*

*val(x : o) = dup val x*

*val(x : i) = s dup val x*

*dup 0 = 0*

*dup s u = s s dup u*

*u + 0 = u*

*u + s v = s(u + v)*

*u \* 0 = 0*

*u \* (s v) = u \* v + u*

0

*sort<sub>1</sub>*

*s sort<sub>1</sub>*

0

*s s sort<sub>1</sub>*

*nat*

*s sort<sub>4</sub>*

0

0 | *nat* | *s nat*

?- *solve(s(val(c)) = val(x), S).*

*s val c = val x*

1 [*dup val x<sub>19</sub> = s val c*]  $\leftarrow$  [*x := x<sub>19</sub> : o*]

2 [*dup val x<sub>23</sub> = val c*]  $\leftarrow$  [*x := x<sub>23</sub> : i*]

**ind**

*s val nil = val x*

*s 0 = val x*

*dup val x<sub>27</sub> = 0*

*0 = val x<sub>27</sub>*

*0 = val x<sub>27</sub>*

*dup val x<sub>27</sub> = 0*

*s 0 = val x*

*s val nil = val x*

*s val(c<sub>25</sub> : o) = val x*

*s dup val c<sub>25</sub> = val x*

*s dup val c<sub>25</sub> = val x*

*s val(c<sub>25</sub> : o) = val x*

*s val(c<sub>26</sub> : i) = val x*

*s s dup val c<sub>26</sub> = val x*

*dup val x<sub>38</sub> = s s dup val c<sub>26</sub>*

*val f<sub>24</sub>(c<sub>26</sub>) = val x<sub>38</sub>*

*val f<sub>24</sub>(c<sub>26</sub>) = val x<sub>38</sub>*

*dup val x<sub>38</sub> = s s dup val c<sub>26</sub>*

*s s dup val c<sub>26</sub> = val x*

*s val(c<sub>26</sub> : i) = val x*

*s val c = val x*

$\leftarrow$  [*x<sub>27</sub> := nil*]

$\leftarrow$  [*x<sub>27</sub> := nil*]

$\leftarrow$  [*x := nil : i*]

$\leftarrow$  [*x := nil : i*]

$\leftarrow$  [*x := c<sub>25</sub> : i*]

$\leftarrow$  [*x := c<sub>25</sub> : i*]

$\leftarrow$  [*x<sub>38</sub> := f<sub>24</sub>(c<sub>26</sub>)*]

$\leftarrow$  [*x<sub>38</sub> := f<sub>24</sub>(c<sub>26</sub>)*]

$\leftarrow$  [*x := f<sub>24</sub>(c<sub>26</sub>) : o*]

$\leftarrow$  [*x := f<sub>24</sub>(c<sub>26</sub>) : o*]

$\leftarrow$  [*x := f<sub>24</sub>(c)*]

*S = [x := f<sub>24</sub>(c)]*

?- *solve(0 + a = a, S).*

*S = []*

?- *solve(s(a) + b = s(a + b), S).*

*S = []*

?- *solve(dup(a) + dup(b) = dup(a + b), S).*

*S = []*

?- solve(val(c) + val(d) = val(x), S).

val c + val d = val x

- 1 [0 = val c + val d]  $\leftarrow$  [x := nil]
- 2 [dup val x<sub>141</sub> = val c + val d]  $\leftarrow$  [x := x<sub>141</sub> : o]
- 3 [s dup val x<sub>153</sub> = val c + val d]  $\leftarrow$  [x := x<sub>153</sub> : i]
- 4 [0 = val d]  $\leftarrow$  [u<sub>162</sub> := val x, x := c]
- 5 [s(u<sub>186</sub> + v<sub>187</sub>) = val x, s v<sub>187</sub> = val d]  $\leftarrow$  [u<sub>186</sub> := val c]

**ind**

val nil + val nil = val x

0 + val nil = val x

0 + 0 = val x

0 = val x

0 = val x

0 + 0 = val x

0 + val nil = val x

val nil + val nil = val x

val nil + val(d<sub>191</sub> : o) = val x

0 + val(d<sub>191</sub> : o) = val x

0 + dup val d<sub>191</sub> = val x

dup val d<sub>191</sub> = val x

dup val d<sub>191</sub> = val x

0 + dup val d<sub>191</sub> = val x

0 + val(d<sub>191</sub> : o) = val x

val nil + val(d<sub>191</sub> : o) = val x

val nil + val(d<sub>192</sub> : i) = val x

0 + val(d<sub>192</sub> : i) = val x

0 + s dup val d<sub>192</sub> = val x

s(0 + dup val d<sub>192</sub>) = val x

s dup val d<sub>192</sub> = val x

s dup val d<sub>192</sub> = val x

s(0 + dup val d<sub>192</sub>) = val x

0 + s dup val d<sub>192</sub> = val x

0 + val(d<sub>192</sub> : i) = val x

val nil + val(d<sub>192</sub> : i) = val x

val(c<sub>189</sub> : o) + val nil = val x

val(c<sub>189</sub> : o) + 0 = val x

dup val c<sub>189</sub> + 0 = val x

dup val c<sub>189</sub> = val x

dup val c<sub>189</sub> = val x

dup val c<sub>189</sub> + 0 = val x

val(c<sub>189</sub> : o) + 0 = val x

val(c<sub>189</sub> : o) + val nil = val x

val(c<sub>189</sub> : o) + val(d<sub>191</sub> : o) = val x

dup val c<sub>189</sub> + val(d<sub>191</sub> : o) = val x

dup val c<sub>189</sub> + dup val d<sub>191</sub> = val x

dup(val c<sub>189</sub> + val d<sub>191</sub>) = val x

dup val f<sub>188</sub>(c<sub>189</sub>, d<sub>191</sub>) = val x

dup val f<sub>188</sub>(c<sub>189</sub>, d<sub>191</sub>) = val x

dup(val c<sub>189</sub> + val d<sub>191</sub>) = val x

dup val c<sub>189</sub> + dup val d<sub>191</sub> = val x

$\leftarrow$  [x := nil]

$\leftarrow$  [x := nil]

$\leftarrow$  [x := nil]

$\leftarrow$  [x := nil]

$\leftarrow$  [x := d<sub>191</sub> : o]

$\leftarrow$  [x := d<sub>191</sub> : o]

$\leftarrow$  [x := d<sub>191</sub> : o]

$\leftarrow$  [x := d<sub>191</sub> : o]

$\leftarrow$  [x := d<sub>192</sub> : i]

$\leftarrow$  [x := d<sub>192</sub> : i]

$\leftarrow$  [x := d<sub>192</sub> : i]

$\leftarrow$  [x := d<sub>192</sub> : i]

$\leftarrow$  [x := d<sub>192</sub> : i]

$\leftarrow$  [x := c<sub>189</sub> : o]

$\leftarrow$  [x := c<sub>189</sub> : o]

$\leftarrow$  [x := c<sub>189</sub> : o]

$\leftarrow$  [x := c<sub>189</sub> : o]

$\leftarrow$  [x := f<sub>188</sub>(c<sub>189</sub>, d<sub>191</sub>) : o]

$\leftarrow$  [x := f<sub>188</sub>(c<sub>189</sub>, d<sub>191</sub>) : o]

$\leftarrow$  [x := f<sub>188</sub>(c<sub>189</sub>, d<sub>191</sub>) : o]



$val\ c + val\ d = val\ x$ 
 $\leftarrow [x := f_{188}(c, d)]$

$S = [x := f_{188}(c, d)]$

$?- solve(a + b + b = a + dup(b), S).$   
 $S = []$

$?- solve(a * dup(b) = dup(a * b), S).$   
 $S = []$

$?- solve(dup(val(c)) + val(d) = val(f_{188}(c : o, d)), S).$   
 $S = []$

$?- solve(val(c) * val(d) = val(x), S).$   
 $(val\ c) * (val\ d) = val\ x$

$1\ [0 = (val\ c) * (val\ d)] \leftarrow [x := nil]$   
 $2\ [dup\ val\ x_{412} = (val\ c) * (val\ d)] \leftarrow [x := x_{412} : o]$   
 $3\ [s\ dup\ val\ x_{419} = (val\ c) * (val\ d)] \leftarrow [x := x_{419} : i]$   
 $4\ [0 = val\ x, 0 = val\ d] \leftarrow [u_{420} := val\ c]$   
 $5\ [u_{427} * v_{428} + u_{427} = val\ x, s\ v_{428} = val\ d] \leftarrow [u_{427} := val\ c]$

$ind[d].$

$(val\ c) * (val\ nil) = val\ x$   
 $(val\ c) * 0 = val\ x$   
 $0 = val\ x$   
 $0 = val\ x$   
 $(val\ c) * 0 = val\ x$   
 $(val\ c) * (val\ nil) = val\ x$   
 $(val\ c) * (val(d_{430} : o)) = val\ x$   
 $(val\ c) * (dup\ val\ d_{430}) = val\ x$   
 $dup(val\ c) * (val\ d_{430}) = val\ x$   
 $dup\ val\ f_{429}(c, d_{430}) = val\ x$   
 $dup\ val\ f_{429}(c, d_{430}) = val\ x$   
 $dup(val\ c) * (val\ d_{430}) = val\ x$   
 $(val\ c) * (dup\ val\ d_{430}) = val\ x$   
 $(val\ c) * (val(d_{430} : o)) = val\ x$   
 $(val\ c) * (val(d_{431} : i)) = val\ x$   
 $(val\ c) * (s\ dup\ val\ d_{431}) = val\ x$   
 $(val\ c) * (dup\ val\ d_{431}) + val\ c = val\ x$   
 $dup(val\ c) * (val\ d_{431}) + val\ c = val\ x$   
 $dup\ val\ f_{429}(c, d_{431}) + val\ c = val\ x$   
 $val\ f_{188}(f_{429}(c, d_{431}) : o, c) = val\ x$   
 $val\ f_{188}(f_{429}(c, d_{431}) : o, c) = val\ x$   
 $dup\ val\ f_{429}(c, d_{431}) + val\ c = val\ x$   
 $dup(val\ c) * (val\ d_{431}) + val\ c = val\ x$   
 $(val\ c) * (dup\ val\ d_{431}) + val\ c = val\ x$   
 $(val\ c) * (s\ dup\ val\ d_{431}) = val\ x$   
 $(val\ c) * (val(d_{431} : i)) = val\ x$   
 $(val\ c) * (val\ d) = val\ x$

$\leftarrow [x := nil]$   
 $\leftarrow [x := nil]$   
 $\leftarrow [x := nil]$   
 $\leftarrow [x := f_{429}(c, d_{430}) : o]$   
 $\leftarrow [x := f_{429}(c, d_{430}) : o]$   
 $\leftarrow [x := f_{429}(c, d_{430}) : o]$   
 $\leftarrow [x := f_{429}(c, d_{430}) : o]$   
 $\leftarrow [x := f_{188}(f_{429}(c, d_{431}) : o, c)]$   
 $\leftarrow [x := f_{188}(f_{429}(c, d_{431}) : o, c)]$   
 $\leftarrow [x := f_{188}(f_{429}(c, d_{431}) : o, c)]$   
 $\leftarrow [x := f_{188}(f_{429}(c, d_{431}) : o, c)]$   
 $\leftarrow [x := f_{188}(f_{429}(c, d_{431}) : o, c)]$   
 $\leftarrow [x := f_{188}(f_{429}(c, d_{431}) : o, c)]$   
 $\leftarrow [x := f_{429}(c, d)]$

$S = [x := f_{429}(c, d)]$

```

?- eval_term(f429(nil : i : o : i, nil : i : i : o), T).
T = nil : i : i : i : i : o

?- listing(≐), listing(law), listing(def).

nat ≐ 0 | s nat.
bin ≐ nil | bin : o | bin : i.
sort1 ≐ 0 | s s sort1.

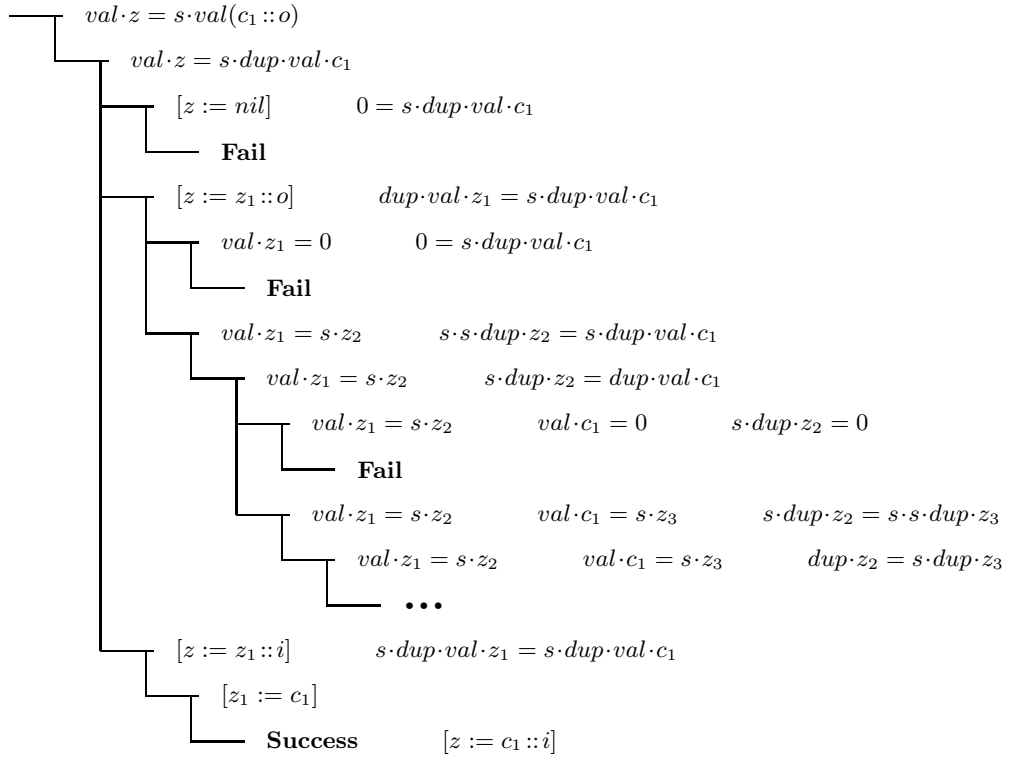
law s val v43 = val f24(v43).
law 0 + v50 = v50.
law s v87 + v88 = s(v87 + v88).
law dup v117 + dup v118 = dup(v117 + v118).
law val v254 + val v255 = val f188(v254, v255).
law v340 + v341 + v341 = v340 + dup v341.
law v356 * (dup v357) = dup v356 * v357.
law dup val v397 + val v398 = val f188(v397 : o, v398).
law (val v446) * (val v447) = val f429(v446, v447).

def val nil = 0.
def val(x : o) = dup val x.
def val(x : i) = s dup val x.
def dup 0 = 0.
def dup s u = s s dup u.
def u + 0 = u.
def u + s v = s(u + v).
def u * 0 = 0.
def u * (s v) = u * v + u.
def f24(nil) = nil : i.
def f24(v35 : o) = v35 : i.
def f24(v42 : i) = f24(v42) : o.
def f188(nil, nil) = nil.
def f188(nil, v201 : o) = v201 : o.
def f188(nil, v207 : i) = v207 : i.
def f188(v215 : o, nil) = v215 : o.
def f188(v224 : o, v225 : o) = f188(v224, v225) : o.
def f188(v231 : o, v232 : i) = f188(v231, v232) : i.
def f188(v238 : i, nil) = v238 : i.
def f188(v244 : i, v245 : o) = f188(v244, v245) : i.
def f188(v252 : i, v253 : i) = f24(f188(v252, v253)) : o.
def f429(v433, nil) = nil.
def f429(v442, v443 : o) = f429(v442, v443) : o.
def f429(v444, v445 : i) = f188(f429(v444, v445) : o, v444).

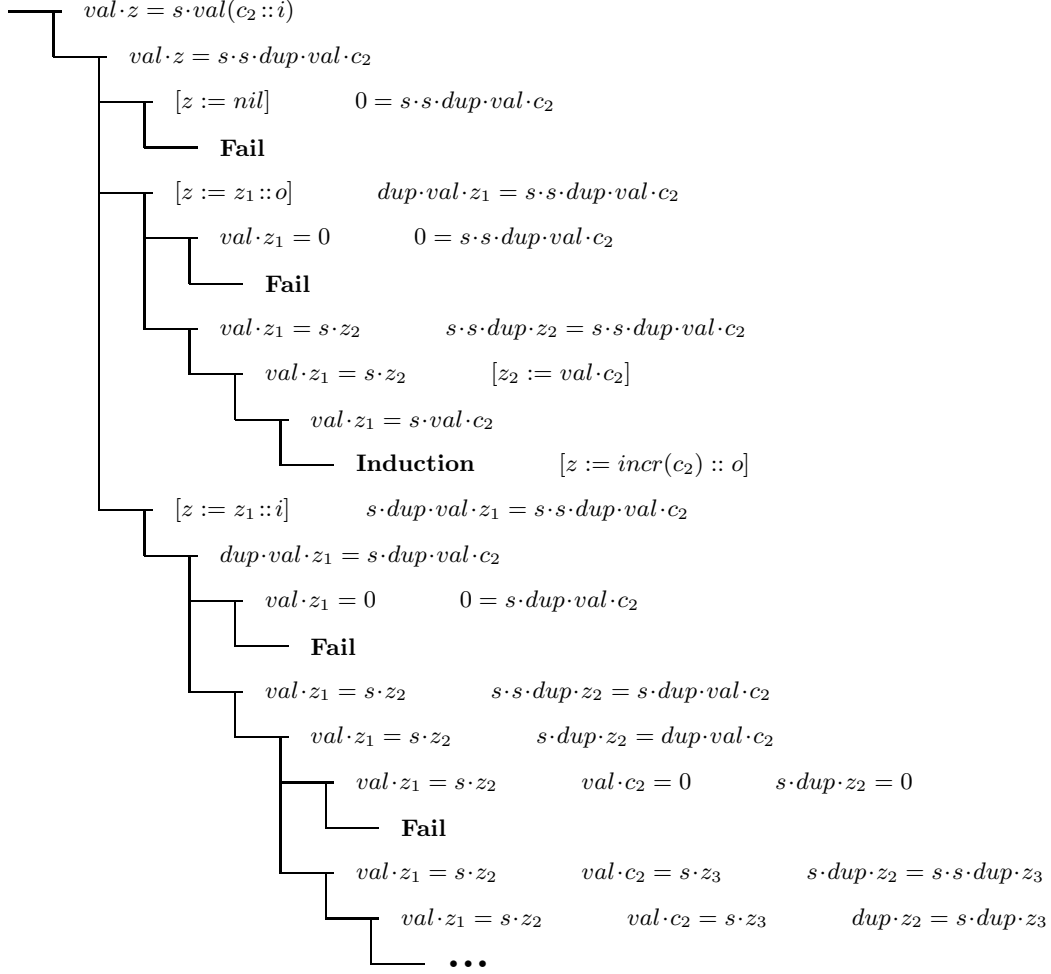
```



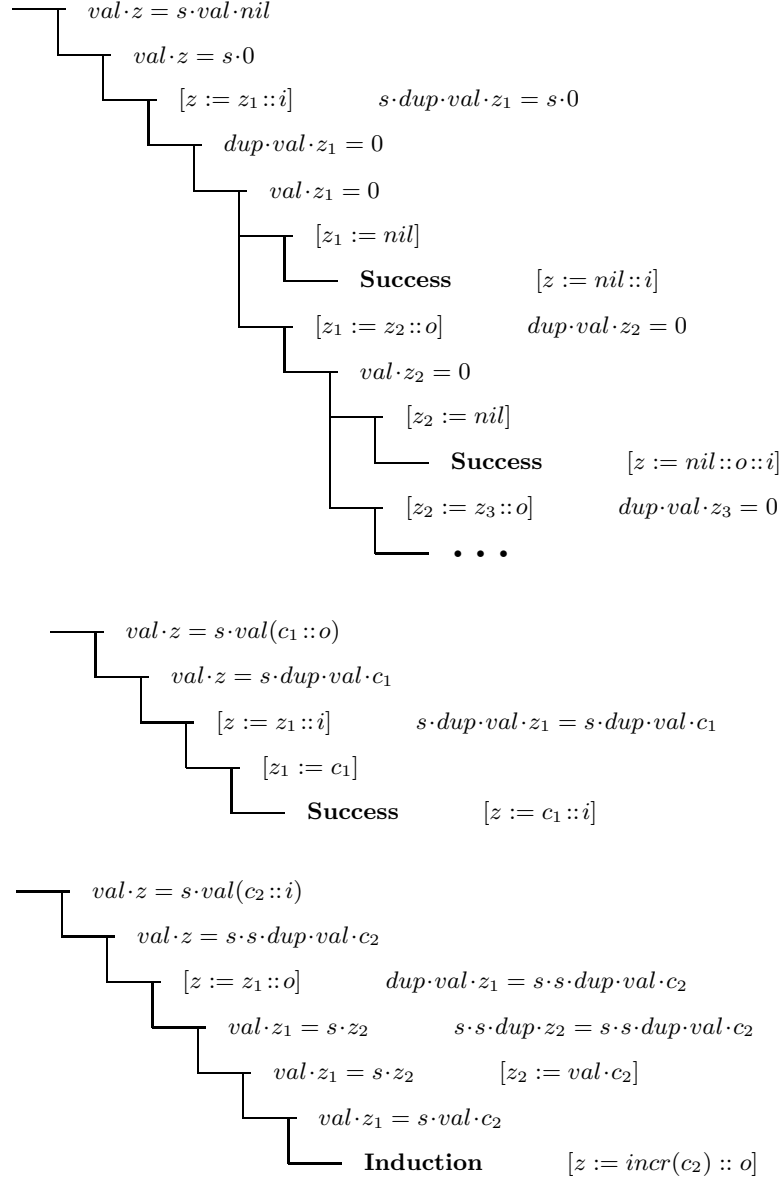




**Fig. 19.** Brute-force search space for equation  $val.z = s.val(c_1 :: o)$



**Fig. 20.** Brute-force search space for equation  $val.z = s.val(c_2 :: i)$



**Fig. 21.** Search space of *incr* synthesis using sorts

## B Case Study “Comb Vector Construction”

In this section, we demonstrate the use of the sort discipline by applying it in a paper case study from the area of compiler construction. The parser generating system PGS is a tool for generating a syntax analyzer for a programming language or, in general, any structured input [10]. The user of PGS has to specify the language to be analyzed by a grammar. The main applications of PGS are in the area of compiler construction, e.g. parsing, syntax analysis or syntax-directed translation.

PGS uses a comb vector technique to compress the two-dimensional array representation of parse tables. A parse table can be merged into one array, called *cont*, where the beginning of each original row is indicated by an entry in an additional array called *base*. In order to be able to distinguish between error and non-error entries, an array called *row* is introduced in parallel to *cont*, containing the row number from which the associated entry in the *cont* array originated.

Given a two-dimensional array, one way of constructing a comb vector is to enter each row into a search tree, lexicographically sorted by the list of its distances. The tree is a son-brother tree, a vertical link pointing to the first son of a node, a horizontal link to the next brother. There are two kinds of nodes, depending on whether a vertical link is necessary or not. A node which has a vertical link corresponds to a distance; brother nodes of this kind are in ascending order with respect to it. A node without a vertical link corresponds to a row number (shown in italics in Fig. 22).

The tree is then traversed in post order, and the corresponding rows are entered into the comb vector. Figure 22 shows an example two-dimensional array together with the constructed search tree. For example, the path *down, right, down, down* corresponds to the distance list 1, 3, 0 of row 4. Figure 23 shows the constructed comb vector and its access function.

Our aim is to define the data structure of a search tree and to construct an algorithm for inserting a list of distances into a search tree. For the sake of simplicity, we do not distinguish between distances and row numbers, representing both by natural numbers.

Assume the constructors

for search trees:	$nil_t$	empty search tree,
	$node1(\cdot, \cdot, \cdot)$	node with vertical and horizontal link,
	$node2(\cdot, \cdot)$	node with horizontal link only,
for distance lists:	$nil_l$	empty list,
	$(\cdot) + (\cdot)$	list “cons”,
for sets of distance lists:	$mt$	empty set, and
	$add(\cdot, \cdot)$	add an element

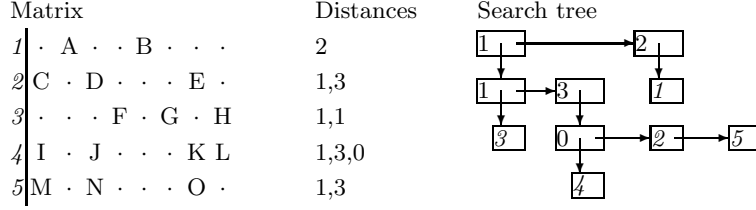
The search tree in Fig. 22 is represented by the term

$$\begin{aligned}
 &node1(1, node1(1, node2(3, nil_t), \\
 &\quad node1(3, node1(0, node2(4, nil_t), \\
 &\quad\quad node2(2, node2(5, nil_t))), \\
 &\quad\quad\quad nil_t)), \\
 &node1(2, node2(1, nil_t), \\
 &\quad\quad nil_t));
 \end{aligned}$$

its set of distance lists can be represented by

$$add(1+1+3+nil_l, add(1+3+0+4+nil_l, add(1+3+2+nil_l, add(1+3+5+nil_l, add(2+1+nil_l, mt)))).$$

To form a valid search tree, a term has to satisfy the following conditions:



**Fig. 22.** Search-tree construction for comb vectors

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	...
row	3	4	3	4	3	.	.	4	4	2	5	2	5	.	.	2	5	
cont	F	I	G	J	H	.	.	K	L	C	M	D	N	.	.	E	O	

	1	2	3	4	5
base	+16	+9	-3	+1	+10

```

get(i,j) = if    row[base[i] + j] == i
           then cont[base[i] + j]
           else 0
           fi

```

**Fig. 23.** Comb vector and access function

- a vertical link may not be  $nil_t$  (38,39),
- the horizontal link of a  $node2$  never points to a  $node1$  (39),
- each horizontal chain of  $node1$ s is in ascending order (first line of 38).

This leads to the sort definitions:

$$Tree \doteq nil_t \mid Tree1 \mid Tree2 \quad (37)$$

$$Tree1 \doteq node1(n:Nat, t_1:Tree1 \mid Tree2, t_2:Tree1) \triangleleft n < t_2 \quad (38)$$

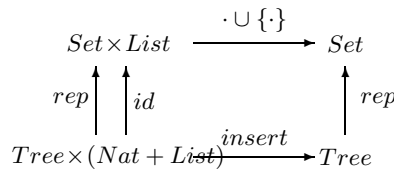
$$\mid node1(n:Nat, t_1:Tree1 \mid Tree2, t_2:Tree2 \mid nil_t)$$

$$Tree2 \doteq node2(Nat, Tree2 \mid nil_t) \quad (39)$$

$$List \doteq nil_l \mid Nat + List \quad (40)$$

$$Set \doteq mt \mid add(List, Set) \quad (41)$$

The definition of  $Tree1$  makes use of a constraint predicate, cf. the remarks at the end of Sect. 2.  $Tree1$  and  $Tree2$  denote the sort of all search trees starting with a  $node1$  and a  $node2$ , respectively. The constraint predicate is defined by the axiom  $n_1 < node1(n_2, t_3, t_4) \leftrightarrow n_1 < n_2$ .



**Fig. 24.** Specification of the insert algorithm

$rep : Tree \rightarrow Set$  gives the set of distance lists represented by a (sub)tree:

$$rep(nil_t) = mt \quad (42)$$

$$rep(node1(n, t_1, t_2)) = n \oplus rep \cdot t_1 \cup rep \cdot t_2 \quad (43)$$

$$rep(node2(n, t)) = add(n + nil_l, rep \cdot t) \quad (44)$$

$\oplus : Nat \times Set \rightarrow Set$  pointwise prefixes a set of distance lists by a new distance:

$$n \oplus mt = mt \quad (45)$$

$$n \oplus add(l, s) = add(n + l, n \oplus s) \quad (46)$$

$\cup : Set \times Set \rightarrow Set$  is the ordinary set union:

$$mt \cup s = s \quad (47)$$

$$add(l, s_1) \cup s_2 = add(l, s_1 \cup s_2) \quad (48)$$

We have the following equations between constructors:

$$add(l_1, add(l_2, s)) = add(l_2, add(l_1, s)) \quad (49)$$

$$add(l, add(l, s)) = add(l, s) \quad (50)$$

Finally, we need the following derived lemma:

$$s_1 \cup s_2 = s_2 \cup s_1 \quad (51)$$

**Fig. 25.** Auxiliary function definitions for search-tree specification

Using the terminology introduced in Sect. 7, we have  $as_1 = Set \times (Nat + List)$ ,  $as_2 = Set$ ,  $ao(s, l) = add(l, s)$ ,  $cs_1 = Tree \times List$ ,  $cs_2 = Tree$ ,  $co(t, l) = insert(t, l)$  is to be synthesized,  $r_1(t, l) = \langle rep(t), l \rangle$ , and  $r_2(t) = rep(t)$ .

The specification uses several auxiliary functions defined in Fig. 25. Expressed in informal terms, it says: “Given a tree  $t$  and a non-empty distance list  $l$ , find a tree  $T$  that contains the same distance lists as  $t$  and additional  $l$ ”; and in formal terms:  $\forall t \in Tree^M, l \in (Nat + List)^M \exists T \in Tree^M \quad rep T = add(l, rep \cdot t)$ . The  $insert$  function will be synthesized as Skolem function for  $T$ .

Using Alg. 73, we obtain the following range sorts of  $rep$ , cf. Fig. 26:

$$\begin{aligned} rg([x := nil_t], rep \cdot x) &= mt \\ rg([x := Tree1], rep \cdot x) &= Sort_{52} \\ rg([x := Tree2], rep \cdot x) &= Sort_{53} \end{aligned}$$

Since the data type  $Set$  is built up from unfree constructors (cf. Eqns. (49) and (50)), we have to somehow compute the normal form sorts, cf. the remarks at the end of Sect. 6. Setting  $Sort_{54} = nf_c(Sort_{52})$  and  $Sort_{55} = nf_c(Sort_{53})$ , we may get:

$$\begin{aligned} Sort_{52} &\doteq add(Nat + Nat + List, Set) \\ Sort_{53} &\doteq add(Nat + nil_l, mt) \mid add(Nat + nil_l, Sort_{53}) \\ Sort_{54} &\doteq add(Nat + Nat + List, Set) \mid add(List, Sort_{54}) \\ Sort_{55} &\doteq Sort_{53} \end{aligned}$$

Intuitively, a term of sort  $Sort_{54}$  denotes a set of distance sequences of which at least one has a length  $\geq 2$ , while a term of sort  $Sort_{55}$  denotes a set of distance sequences of length 1.  $Sort_{52}$ ,  $Sort_{53}$ , and  $mt$  are pairwise disjoint, as are  $Sort_{54}$ ,  $Sort_{55}$ , and  $mt$ . These signatures are too complex for there to be much likelihood of their being declared by a user who does not know the proof in advance. The estimation of range sorts, especially of  $rep$ , with such precision that inputs starting with different constructors result in disjoint output sorts is the main contribution of the sort discipline to search-space reduction in this example.

When verifying by hand, without use of the sort discipline, some intuition *is* needed to find out which values  $rep(node1(n, t_1, t_2))$  can have:

First, we always have  $rep(node2(n, t)) = add(n+nil_l, rep(t)) \neq mt$ .

Then,  $rep(node1(n, t_1, t_2)) = n \oplus rep(t_1) \cup rep(t_2)$ , where  $t_2$  may be  $nil_t$  and thus  $rep(t_2) = mt$ , but  $t_1$  has again the form  $node1(n', t'_1, t'_2)$ <sup>9</sup> and thus (by I.H.)  $rep(t_1) \neq mt$ , hence also  $n \oplus rep(t_1) \neq mt$ . Thus, we always have  $rep(node1(n, t_1, t_2)) \neq mt$ .

Finally,  $rep(t_1)$  contains at least one distance sequence of length  $\geq 1$  (for  $t_1 = node2(n', t')$  trivial, for  $t_1 = node1(n', t'_1, t'_2)$  by I.H.); that is why  $n \oplus rep(t_1) \subset rep(node1(n, t_1, t_2))$  has to contain at least one distance sequence of length  $\geq 2$ .

The “intuition” in this argumentation consists in recognizing two induction hypotheses and verifying them as valid. The main difficulty here consists in recognizing suitable hypotheses; checking of their validity could probably be carried out by an arbitrary induction prover. It is precisely this task of recognition that is performed by the sort discipline. The two implicitly made inductions in the intuitive argumentation correspond to applications of the global transformation rules from Lemmas 68 and 69, cf. Fig. 26.

Figures 28 to 35 show the synthesis proof. Variables are denoted by upper-case letters, constants by lower-case letters. A number in the right-hand column refers to the equation that has been used for narrowing (rule (ln) in Thm. 75), an exponent “-” denoting the reversed equation; “dec” and “I.H.” mean the application of the decomposition rule (rule (d) in Thm. 75), and the induction hypothesis, respectively. Narrowing steps that are *not* uniquely determined by the sort discipline are marked with “\*”. They all occur as a series of backward applications of laws for  $\oplus$  or  $\cup$  in order to get the right-hand side close to the syntactic structure of the left-hand side and then perform a decomposition. Application of the induction hypothesis is marked with “(\*)” since it need only be taken into account if the actual equation’s sort is too large to determine a narrowing step uniquely.

In cases that do not use the induction hypothesis, the sort restrictions enable us to find the solution automatically. For example, in case 2.1 ( $t = node2(n_1, t_2)$ ,  $l = n_4 + nil_l$ , cf. Fig. 30), owing to sort restrictions, only Eqn. (44) can be used in the narrowing step, since the right-hand side has the sort  $add(Nat+nil_l, Sort_{53}) \subset Sort_{53}$ . In cases that actually use the induction hypothesis, the sort restrictions prune the search space to the size of a verification proof, solving the additional problems of synthesis. Moreover, the use of the sort discipline allows us to perform the crucial proper narrowing step as the very first one, providing syntactic information at the equations’ left-hand side, which can be used by subsequent steps concerned with E-unification wrt. the *Set* equations.

The breaking-down of case 3.2 ( $t = node1(n_1, t_2, t_3)$ ,  $l = n_4 + n_5 + l_6$ ) into three subcases 3.2.1 – 3.2.3 can be done automatically. Each solution has to fulfill the additional requirement that  $insert(\dots) \in Tree^M$ , since this is not ensured by the narrowing process itself. After having found the solution  $T = node1(n_4, insert(nil_t, n_5 + l_6), t)$  shown in Fig. 33, it can be determined that  $T \in Tree^M$  only if  $insert(nil_t, n_5 + l_6) \in Tree1^M \cup Tree2^M$  and  $n_4 < n_1$ . The former condition is delayed until the synthesis of the algorithm is complete and can then be verified by an easy induction. The latter condition is intended to be passed to a prover in which the sort algorithms are embedded; it must be able to detect that  $n_4 < n_1 \not\Rightarrow true$  and to initiate the search for further solutions of case 3.2. In this way, the solutions shown in Figs. 34 and 35 are found. Finally, the prover must be able to detect that all subcases have been covered, i.e.  $n_4 < n_1 \vee n_1 < n_4 \vee n_4 = n_1 \Leftrightarrow true$ .

The synthesized algorithm is shown in Fig. 27.

<sup>9</sup> Or  $node2(n', t')$ , see above.



$$\begin{aligned}
& (Tree1:x) (rep.x:z) \\
= & (Nat:n_1) (Tree1:t_1) (Tree:t_2) (n_1 \oplus rep.t_1 \cup rep.t_2 : z) \\
& | (Nat:n_1) (Tree2:t_1) (Tree:t_2) (n_1 \oplus rep.t_1 \cup rep.t_2 : z) \\
= & \dots \\
= & (Nat:n_1) (Tree1:t_1) (Tree:t_2) (rep.t_1 : mt) (mt \cup rep.t_2 : z) \\
& | (Nat:n_1) (Tree1:t_1) (Tree:t_2) (rep.t_1 : add(l_1, s_1)) (add(n_1+l_1, n_1 \oplus s_1 \cup rep.t_2) : z) \\
& | (Nat:n_1) (Tree2:t_1) (Tree:t_2) (add(Nat+Nat+nil_l, Set) : z) \\
= & (Nat:n_1) (Tree1:t_1) (Tree:t_2) (rep.t_1 : add(l_1, s_1)) (add(n_1+l_1, n_1 \oplus s_1 \cup rep.t_2) : z) \\
& | (Nat:n_1) (Tree2:t_1) (Tree:t_2) (add(Nat+Nat+nil_l, Set) : z) \quad (*) \\
\stackrel{(\subset)}{=} & (Nat:n_1) (Tree1:t_1) (Tree:t_2) (rep.t_1 : add(l_1, s_1)) (add(n_1+l_1, max_\cup) : z) \quad (**) \\
& | (Nat:n_1) (Tree2:t_1) (Tree:t_2) (add(Nat+Nat+nil_l, Set) : z) \\
= & (Nat:n_1) (Tree1:t_1) (Tree:t_2) (rep.t_1 : add(l_1, s_1)) (add(n_1+l_1, max_\cup) : z) (l_1 : Sort_{56}) \\
& | (Nat:n_1) (Tree2:t_1) (Tree:t_2) (add(Nat+Nat+nil_l, Set) : z) \quad (***) \\
\stackrel{(\subset)}{=} & (Nat:n_1) (Tree1:t_1) (Tree:t_2) (add(n_1+l_1, max_\cup) : z) (l_1 : Sort_{56}) \\
& | (Nat:n_1) (Tree2:t_1) (Tree:t_2) (add(Nat+Nat+nil_l, Set) : z)
\end{aligned}$$

(\*): Constructor term deletion (cf. Lemma 68): the first alternative can be removed since  $rep.x$  does not produce  $mt$  outside of it, but it requires, in turn, the production of  $mt$  by  $rep.t_1$ . Note that the constructors  $mt$  and  $add(\cdot, \cdot)$  are regarded as free such that  $add(x, y) \neq mt$  always holds. It has been shown that it is sufficient to consider Eqns. (49) and (50) only outside the  $rg$  computation.

(\*\*): Estimation by trivial upper bound, cf. Alg. 70, and (\*\*) in Fig. 14;  $max_\cup = Set$ .

(\*\*\*): Constructor argument estimation (cf. Lemma 69): if  $rep.x$  yields  $add(l_1, s_1)$ ,  $l_1$  has the form  $Nat+Nat+nil_l$  from the second alternative or  $add(n'_1+l'_1, \dots)$  from the first one, where the same holds, in turn, for  $l'_1$ .

Hence,  $l_1$  belongs to  $Sort_{56}$ , where  $Sort_{56} \doteq Nat+Nat+nil_l \mid Nat+Sort_{56}$ .

The result we get is  $(Tree1:x) (rep.x:z) = (Sort_{52}:z)$   
where  $Sort_{52} \doteq add(Nat+Sort_{56}, Set) \mid add(Nat+Nat+nil_l, Set)$ ,  
i.e.  $Sort_{52} = add(Nat+Nat+List, Set)$ .

**Fig. 26.** Range sort computation for  $rep$

$$\begin{aligned}
insert(nil_t, n_4+nil_l) &= node2(n_4, nil_t) \\
insert(nil_t, n_4+n_5+l_6) &= node1(n_4, insert(nil_t, n_5+l_6), nil_t) \\
insert(node2(n_1, t_2), n_4+nil_l) &= node2(n_4, node2(n_1, t_2)) \\
insert(node2(n_1, t_2), n_4+n_5+l_6) &= node1(n_4, insert(nil_t, n_5+l_6), node2(n_1, t_2)) \\
insert(node1(n_1, t_2, t_3), n_4+nil_l) &= node1(n_1, t_2, insert(t_3, n_4+nil_l)) \\
insert(node1(n_1, t_2, t_3), n_4+n_5+l_6) &= node1(n_4, insert(nil_t, n_5+l_6), node1(n_1, t_2, t_3)) \\
&\quad \leftarrow n_4 < n_1 \\
insert(node1(n_1, t_2, t_3), n_4+n_5+l_6) &= node1(n_1, t_2, insert(t_3, n_4+n_5+l_6)) \leftarrow n_1 < n_4 \\
insert(node1(n_1, t_2, t_3), n_4+n_5+l_6) &= node1(n_1, insert(t_2, n_5+l_6), t_3) \leftarrow n_4 = n_1
\end{aligned}$$

**Fig. 27.** Synthesized comb vector insertion algorithm

$rep \cdot T = add(n_4 + nil_l, rep \cdot nil_t)$	
$add(N_7 + nil_l, rep \cdot T_8) = add(n_4 + nil_l, rep \cdot nil_t) \wedge$ $T = node2(N_7, T_8)$	(44)
$add(N_7 + nil_l, rep \cdot T_8) = add(n_4 + nil_l, mt)$	(42)
$N_7 = n_4 \wedge$ $rep \cdot T_8 = mt$	dec.
$T_8 = nil_t$	(42)

Answer substitution:  $[N_7 := n_4, T_8 := nil_t]$   
 $\circ [T := node2(N_7, T_8)]$

**Fig. 28.** Case 1.1 —  $t = nil_t, l = n_4 + nil_l$

$rep \cdot T = add(n_4 + n_5 + l_6, rep \cdot nil_t)$		
$N_7 \oplus rep \cdot T_8 \cup rep \cdot T_9 = add(n_4 + n_5 + l_6, rep \cdot nil_t) \wedge$ $T = node1(N_7, T_8, T_9)$	(43)	
$N_7 \oplus rep \cdot T_8 \cup rep \cdot T_9 = add(n_4 + n_5 + l_6, mt \cup rep \cdot nil_t)$	(47) <sup>-</sup>	*
$N_7 \oplus rep \cdot T_8 \cup rep \cdot T_9 = add(n_4 + n_5 + l_6, mt) \cup rep \cdot nil_t$	(48) <sup>-</sup>	*
$N_7 \oplus rep \cdot T_8 \cup rep \cdot T_9 = add(n_4 + n_5 + l_6, n_4 \oplus mt) \cup rep \cdot nil_t$	(45) <sup>-</sup>	*
$N_7 \oplus rep \cdot T_8 \cup rep \cdot T_9 = n_4 \oplus add(n_5 + l_6, mt) \cup rep \cdot nil_t$	(46) <sup>-</sup>	*
$N_7 = n_4 \wedge$ $rep \cdot T_8 = add(n_5 + l_6, mt) \wedge$ $T_9 = nil_t$	dec.	*
$add(L_{11}, rep \cdot T_{10}) = add(n_5 + l_6, mt) \wedge$ $T_8 = insert(T_{10}, L_{11})$	I.H.	(*)
$L_{11} = n_5 + l_6 \wedge$ $rep \cdot T_{10} = mt$	dec.	
$T_{10} = nil_t$	(42)	

Answer substitution:  $[L_{11} := n_5 + l_6, T_{10} := nil_t]$   
 $\circ [N_7 := n_4, T_9 := nil_t, T_8 := insert(T_{10}, L_{11})]$   
 $\circ [T := node1(N_7, T_8, T_9)]$

**Fig. 29.** Case 1.2 —  $t = nil_t, l = n_4 + n_5 + l_6$

$rep.T = add(n_4 + nil_l, rep.node2(n_1, t_2))$	
$add(N_7 + nil_l, rep.T_8) = add(n_4 + nil_l, rep.node2(n_1, t_2)) \wedge$ $T = node2(N_7, T_8)$	(44)
$N_7 = n_4 \wedge$ $T_8 = node2(n_1, t_2)$	dec.

Answer substitution:  $[N_7 := n_4, T_8 := node2(n_1, t_2)]$   
 $\circ [T := node2(N_7, T_8)]$

**Fig. 30.** Case 2.1 —  $t = node2(n_1, t_2)$ ,  $l = n_4 + nil_l$

$rep.T = add(n_4 + n_5 + l_6, rep.node2(n_1, t_2))$	
$N_7 \oplus rep.T_8 \cup rep.T_9 = add(n_4 + n_5 + l_6, rep.node2(n_1, t_2)) \wedge$ $T = node1(N_7, T_8, T_9)$	(43)
$N_7 \oplus rep.T_8 \cup rep.T_9 = add(n_4 + n_5 + l_6, mt \cup rep.node2(n_1, t_2))$	(47) <sup>-</sup> *
$N_7 \oplus rep.T_8 \cup rep.T_9 = add(n_4 + n_5 + l_6, mt) \cup rep.node2(n_1, t_2)$	(48) <sup>-</sup> *
$N_7 \oplus rep.T_8 \cup rep.T_9 = add(n_4 + n_5 + l_6, n_4 \oplus mt) \cup rep.node2(n_1, t_2)$	(45) <sup>-</sup> *
$N_7 \oplus rep.T_8 \cup rep.T_9 = n_4 \oplus add(n_5 + l_6, mt) \cup rep.node2(n_1, t_2)$	(46) <sup>-</sup> *
$N_7 = n_4 \wedge$ $rep.T_8 = add(n_5 + l_6, mt) \wedge$ $T_9 = node2(n_1, t_2)$	dec. *
$add(L_{11}, rep.T_{10}) = add(n_5 + l_6, mt) \wedge$ $T_8 = insert(T_{10}, L_{11})$	I.H. (*)
$L_{11} = n_5 + l_6 \wedge$ $rep.T_{10} = mt$	dec.
$T_{10} = nil_t$	(42)

Answer substitution:  $[L_{11} := n_5 + l_6, T_{10} := nil_t]$   
 $\circ [N_7 := n_4, T_9 := node2(n_1, t_2), T_8 := insert(T_{10}, L_{11})]$   
 $\circ [T := node1(N_7, T_8, T_9)]$

**Fig. 31.** Case 2.2 —  $t = node2(n_1, t_2)$ ,  $l = n_4 + n_5 + l_6$

$rep.T = add(n_4 + nil_l, rep.node1(n_1, t_2, t_3))$	
$N_7 \oplus rep.T_8 \cup rep.T_9 = add(n_4 + nil_l, rep.node1(n_1, t_2, t_3)) \wedge$ $T = node1(N_7, T_8, T_9)$	(43)
$N_7 \oplus rep.T_8 \cup rep.T_9 = add(n_4 + nil_l, n_1 \oplus rep.t_2 \cup rep.t_3)$	(43)
$N_7 \oplus rep.T_8 \cup rep.T_9 = add(n_4 + nil_l, rep.t_3 \cup n_1 \oplus rep.t_2)$	(51) *
$N_7 \oplus rep.T_8 \cup rep.T_9 = add(n_4 + nil_l, rep.t_3) \cup n_1 \oplus rep.t_2$	(48) <sup>-</sup> *
$N_7 \oplus rep.T_8 \cup rep.T_9 = n_1 \oplus rep.t_2 \cup add(n_4 + nil_l, rep.t_3)$	(51) *
$N_7 = n_1 \wedge$ $T_8 = t_2 \wedge$ $rep.T_9 = add(n_4 + nil_l, rep.t_3)$	dec. *
$add(L_{11}, rep.T_{10}) = add(n_4 + nil_l, rep.t_3) \wedge$ $T_9 = insert(T_{10}, L_{11})$	I.H. (*)
$L_{11} = n_4 + nil_l \wedge$ $T_{10} = t_3$	dec.

Answer substitution:  $[L_{11} := n_4 + nil_l, T_{10} := t_3]$   
 $\circ [N_7 := n_1, T_8 := t_2, T_9 := insert(T_{10}, L_{11})]$   
 $\circ [T := node1(N_7, T_8, T_9)]$

**Fig. 32.** Case 3.1 —  $t = node1(n_1, t_2, t_3)$ ,  $l = n_4 + nil_l$

$rep.T = add(n_4+n_5+l_6, rep.node1(n_1, t_2, t_3))$		
$N_7 \oplus rep.T_8 \cup rep.T_9 = add(n_4+n_5+l_6, rep.node1(n_1, t_2, t_3)) \wedge$ $T = node1(N_7, T_8, T_9)$	(43)	
$N_7 \oplus rep.T_8 \cup rep.T_9 = add(n_4+n_5+l_6, mt \cup rep.node1(n_1, t_2, t_3))$	(47) <sup>-</sup>	*
$N_7 \oplus rep.T_8 \cup rep.T_9 = add(n_4+n_5+l_6, mt) \cup rep.node1(n_1, t_2, t_3)$	(48) <sup>-</sup>	*
$N_7 \oplus rep.T_8 \cup rep.T_9 = add(n_4+n_5+l_6, n_4 \oplus mt) \cup rep.node1(n_1, t_2, t_3)$	(45) <sup>-</sup>	*
$N_7 \oplus rep.T_8 \cup rep.T_9 = n_4 \oplus add(n_5+l_6, mt) \cup rep.node1(n_1, t_2, t_3)$	(46) <sup>-</sup>	*
$N_7 = n_4 \wedge$ $rep.T_8 = add(n_5+l_6, mt) \wedge$ $T_9 = node1(n_1, t_2, t_3)$	dec.	*
$add(L_{11}, rep.T_{10}) = add(n_5+l_6, mt) \wedge$ $T_8 = insert(T_{10}, L_{11})$	I.H.	(*)
$L_{11} = n_5+l_6 \wedge$ $rep.T_{10} = mt$	dec.	
$T_{10} = nil_t$	(42)	

Answer substitution:  $[L_{11} := n_5+l_6, T_{10} := nil_t]$   
 $\circ [N_7 := n_4, T_9 := node1(n_1, t_2, t_3), T_8 := insert(T_{10}, L_{11})]$   
 $\circ [T := node1(N_7, T_8, T_9)]$

**Fig. 33.** Case 3.2.1 —  $t = node1(n_1, t_2, t_3)$ ,  $l = n_4+n_5+l_6$ ,  $n_4 < n_1$

$rep.T = add(n_4+n_5+l_6, rep.node1(n_1, t_2, t_3))$		
$N_7 \oplus rep.T_8 \cup rep.T_9 = add(n_4+n_5+l_6, rep.node1(n_1, t_2, t_3)) \wedge$ $T = node1(N_7, T_8, T_9)$	(43)	
$N_7 \oplus rep.T_8 \cup rep.T_9 = add(n_4+n_5+l_6, n_1 \oplus rep.t_2 \cup rep.t_3)$	(43)	
$N_7 \oplus rep.T_8 \cup rep.T_9 = add(n_4+n_5+l_6, rep.t_3 \cup n_1 \oplus rep.t_2)$	(51)	*
$N_7 \oplus rep.T_8 \cup rep.T_9 = add(n_4+n_5+l_6, rep.t_3) \cup n_1 \oplus rep.t_2$	(48) <sup>-</sup>	*
$N_7 \oplus rep.T_8 \cup rep.T_9 = n_1 \oplus rep.t_2 \cup add(n_4+n_5+l_6, rep.t_3)$	(51)	*
$N_7 = n_1 \wedge$ $T_8 = t_2 \wedge$ $rep.T_9 = add(n_4+n_5+l_6, rep.t_3)$	dec.	*
$add(L_{11}, rep.T_{10}) = add(n_4+n_5+l_6, rep.t_3) \wedge$ $T_9 = insert(T_{10}, L_{11})$	I.H.	(*)
$L_{11} = n_4+n_5+l_6 \wedge$ $T_{10} = t_3$	dec.	

Answer substitution:  $[L_{11} := n_r+n_5+l_6, T_{10} := t_3]$   
 $\circ [N_7 := n_1, T_8 := t_2, T_9 := insert(T_{10}, L_{11})]$   
 $\circ [T := node1(N_7, T_8, T_9)]$

**Fig. 34.** Case 3.2.2 —  $t = node1(n_1, t_2, t_3)$ ,  $l = n_4+n_5+l_6$ ,  $n_1 < n_4$

$rep \cdot T = add(n_1+n_5+l_6, rep \cdot node1(n_1, t_2, t_3))$		
$N_7 \oplus rep \cdot T_8 \cup rep \cdot T_9 = add(n_1+n_5+l_6, rep \cdot node1(n_1, t_2, t_3)) \wedge$ $T = node1(N_7, T_8, T_9)$	(43)	
$N_7 \oplus rep \cdot T_8 \cup rep \cdot T_9 = add(n_1+n_5+l_6, n_1 \oplus rep \cdot t_2 \cup rep \cdot t_3)$	(43)	
$N_7 \oplus rep \cdot T_8 \cup rep \cdot T_9 = add(n_1+n_5+l_6, n_1 \oplus rep \cdot t_2) \cup rep \cdot t_3$	(48) <sup>-</sup>	*
$N_7 \oplus rep \cdot T_8 \cup rep \cdot T_9 = n_1 \oplus add(n_5+l_6, rep \cdot t_2) \cup rep \cdot t_3$	(46) <sup>-</sup>	*
$N_7 = n_1 \wedge$ $rep \cdot T_8 = add(n_5+l_6, rep \cdot t_2) \wedge$ $T_9 = t_3$	dec.	*
$add(L_{11}, rep \cdot T_{10}) = add(n_5+l_6, rep \cdot t_2) \wedge$ $T_8 = insert(T_{10}, L_{11})$	I.H.	(*)
$L_{11} = n_5+l_6 \wedge$ $T_{10} = t_2$	dec.	

Answer substitution:  $[L_{11} := n_5+l_6, T_{10} := t_2]$   
 $\circ [N_7 := n_1, T_9 := t_3, T_8 := insert(T_{10}, L_{11})]$   
 $\circ [T := node1(N_7, T_8, T_9)]$

**Fig. 35.** Case 3.2.3 —  $t = node1(n_1, t_2, t_3)$ ,  $l = n_4+n_5+l_6$ ,  $n_4 = n_1$

## C Index

NOTION	Nr.	Page		
			annotated term	56 29
			application	13 15
			application	18 16
			$apply(\sigma, x)$	37 21
			approximation rule	64 32
$\dashv$	77	39	$ar(g)$	1 7
$\rightarrow$	77	39	arity	1 7
$\leftrightarrow^*$	61	30	$ar(\mathbf{cr})$	13 15
$\rightarrow^*$	61	30	$Bin$	4 8
$\sigma_{v_1} \rightarrow \sigma_{v_2}$	61	30	$Bin$	78 39
$\mu_i f(u_{i1}, \dots, u_{in})$	60	30	$compose(\sigma, \tau)$	41 23
$\subseteq$	1	7	computation path	64 32
$\neq$	1	7	computation tree	64 32
$\subset$	1	7	constraint	12 12
$A \times B$	2	7	constructor symbols	1 7
$\doteq$	4	8	constructor terms	1 7
$\ddot{<}$	4	8	constructor-matching rules	9 9
$\dot{<}$	4	8	constructors for t-substitutions	13 15
$\bigcup_{i=1}^n S_i$	2	7	$cr$	1 7
$ $	4	8	$\mathbf{cr}$	13 15
$\perp$	36	21	$\mathbf{cr} \diamond \mathbf{cr}'$	13 15
$\top$	36	21	$\mathbf{cr} _{V'}$	13 15
$\top_V$	36	21	$\mathbf{cr}_x$	13 15
$(V \rightarrow \mathcal{CR})$	13	15	$\mathcal{CR}$	1 7
$(V \hookrightarrow \mathcal{CR})$	17	16	(d)	75 37
$\varepsilon$	13	15	defining equations	59 29
$\beta$	3	7	defining equations	60 30
$\beta _V$	3	7	dependent	38 22
$\beta v$	3	7	depth	3 7
$\beta_1 \circ \beta_2$	3	7	$diff(S_1, S_2 \mid \dots \mid S_m)$	11 12
$\gamma$	3	7	distributivity rules	9 9
$\mu$	15	15	$div(\sigma, \beta)$	55 27
$\mu'$	15	15	$dom(\beta)$	3 7
$\sigma$	15	15	$dom(f)$	60 30
$\sigma'$	15	15	$dom(f, I)$	60 30
$\sigma/\beta$	26	18	$dom(\sigma')$	17 16
$\sigma'/\beta$	26	18	$dom(\sigma)$	36 21
$\sigma \circ \beta$	24	17	$dup(\sigma, \beta)$	44 23
$\sigma' \circ \beta$	24	17	$dup(x)$	78 39
$\sigma \diamond \tau$	22	17	elementwise extension	2 7
$\sigma' \diamond \tau'$	22	17	equations between constructors	77 39
$\sigma _V$	21	17	$Even$	78 39
$\sigma' _V$	21	17	extended sort	57 29
$\sigma u$	18	16	$f$	1 7
$\sigma' u$	18	16	$\mathcal{F}$	1 7
$\tau$	15	15	$f \cdot x$	2 7
$\tau'$	15	15	$f[A']$	2 7
$\sigma_v$	56	29	factorization	26 18
$::$	4	8	$fact(\sigma, \beta)$	47 25
$0_x s_y$	14	15	$fact(\sigma, \beta)$	49 26
$abstract(S, x)$	42	23	$g$	1 7
admissible t-substitutions	15	15	ground constructor terms	1 7
alternatives	64	32		

homogeneous	48	26	$restrict(\sigma, V)$	40	22
$i$	4	8	rewrite relation	61	30
independent	38	22	$rg$	59	29
Induction Principle	6	8	$rg_c$	78	39
$inf(S_1, S_2)$	10	12	$rg(\sigma, v)$	73	35
inhabitation	9	9	$S$	1	7
$inh(S, Occ)$	12	12	$\mathcal{S}$	1	7
intersection	9	9	semantics	4	8
junk terms	61	30	semi-independent	38	22
lazy narrowing	75	37	semilinear	3	7
$Lex_{x < y}$	42	23	$(\sigma)$	62	32
lifting	23	17	$single(S)$	12	12
linear	3	7	$S^M$	4	8
linear	3	7	$snoc$	4	8
(ln)	75	37	solution of an equation	74	37
local transformation rules	64	32	sort definitions	4	8
loop-checking rules	9	9	sort equivalence	9	9
$M$	4	8	sort expressions	4	8
$max_f$	70	34	sort names	1	7
$max'_f$	71	34	sort system	4	8
$mgu(v_1, v_2)$	3	7	Sorted Narrowing	74	37
most general unifier	3	7	Sorted Rewriting	60	30
$Mtch_{x,y}$	42	23	$SortName$	4	8
$Nat$	78	39	subsort	9	9
$Nat_x$	36	21	substitution	3	7
$Nat_{x,y}$	36	21	$Sum_{x,y,z}$	42	23
$Nat_{x < y}$	36	21	$S^X$	4	8
$Nat_{x=y}$	36	21	$\mathcal{T}_{C\mathcal{R}}$	1	7
$Nat_y$	36	21	$\mathcal{T}_{C\mathcal{R}, \mathcal{F}} / \leftrightarrow^*$	61	30
$nf[A]$	61	30	$\mathcal{T}_{C\mathcal{R}, \mathcal{F}, \nu}$	1	7
$nf_c$	78	39	$\mathcal{T}_{C\mathcal{R}, \mathcal{F}, \nu, \mathcal{S}}$	1	7
$nf(v)$	61	30	$\mathcal{T}_{C\mathcal{R}, \nu}$	1	7
non-constructor functions	1	7	$\mathcal{T}_{(V \rightarrow C\mathcal{R})}$	15	15
$o$	4	8	$\mathcal{T}_X$	1	7
ordinary substitution	3	7	$\mathcal{T}_{X,Y}$	1	7
parallel composition	13	15	$\mathcal{T}_{(V \rightarrow C\mathcal{R})}^*$	15	15
parallel composition	22	17	$\mathcal{T}_{(V \hookrightarrow C\mathcal{R})}^*$	17	16
parallel composition	3	7	terms	1	7
partial mappings	17	16	t-sets	15	15
$Pref_{x,y,z}$	42	23	t-substitutions	15	15
pseudolinear	3	7	tuple	2	7
pseudolinear	3	7	$u$	1	7
$ran(\beta)$	3	7	$use(S)$	6	8
Rank of a T-Substitution	64	32	$v$	1	7
$rank(\sigma', (w_1 : u_1))$	65	32	$\mathcal{V}$	1	7
$rank(\sigma', (w_1 : u_1) \dots (w_n : u_n))$	65	32	$v_1 \triangleleft v_2$	3	7
regular	6	8	$\neq$		
relative complement	9	9	$v_1 \triangleleft v_2$	3	7
renaming substitution	3	7	$\langle v_1, \dots, v_n \rangle$	2	7
restriction	13	15	$\langle v_i \mid p(v_i), i = 1, \dots, n \rangle$	2	7
restriction	21	17	$val$	78	39
restriction	3	7	variables	1	7

$vars(v_1, \dots, v_n)$	3	7
$w$	1	7
$((w_1 : u_1) (w_2 : u_2))^M$	62	32
well-defined	61	30
$w^M$	62	32
$(w : u)^M$	62	32
$x$	1	7
$[x := S]$	23	17
$[x := u]$	23	17
$[x_1 := v_1, \dots, x_n := v_n]$	3	7
$x:S$	2	7
$y$	1	7
$z$	1	7
$Zip_{x,y,z}$	42	23